A MEDIUM - OF - INSTRUCTION PROGRAMMING LANGUAGE - MINIPL

(PART I)

By
ARVIND AGRAWAL

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
AUGUST, 1974

A MEDIUM-OF-INSTRUCTION PROGRAMMING LANGUAGE-MINIPL

(PART I)

A Thesis Submitted

In Partial Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

by

Arvind Agrawal

to the

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

August 1974

CERTIFICATE

This is to certify that the thesis entitled, "A Medium-of-Instruction Programming Language-MINIPL" is a record of the work carried out under our supervision and that it has not been submitted elsewhere for a degree.

V. K. Vaishnavi
Lecturer
Computer Science Group
Indian Institute of Technology, Kanpur

August, 1974.

Dr. H. V. Sahasrabuddhe
Assistant Professor
Department of Electrical Engineering
Indian Institute of Technology, Kanpur

ACKNOWLEDGEMENTS

I am deeply indebted to Dr. H.V. Sahasrabuddhe and Mr. V.K. Vaishnavi for their valuable guidance and encouragement from time to time. It will be vain to thank my collegue Mr. S.S. Sibia withoutwhose sustained cooperation the project would not have reached completion.

I thank Messers S.S. Pethkar, G.L. Mishra and S. K. Tewari for their excellent and speedy typing help.

ARVIND AGRAWAL

Kanpur
August 1974.

ABSTRACT

The needs for a medium-of-instruction programming language are identified. Language design criteria are developed in the context of these needs. A language which purports to meet these requirements has been specified.

Basic structure of a mobile compiler for the specified language is implemented using the transition matrix technique for syntax analysis.

CONTENTS

PREFACE

The unsuitability of most of the important modern programming languages as a vehicle to teach important concepts of 'programming' to those who are about to embark on making it their career, has been felt by teachers shouldering the responsibility. This awareness has become more acute with the emergence in the recent years of 'programming' as a major computer science discipline worsthy of serious academic pursuit. The present project, or at least part of it, is a bid to respond to the above need. Its seeks to specify a medium-of-instruction language for a first course in the programming for beginning programmers-the underlined letters supplying us with a name of such a language (MIPL). As we shall see later, the language specified is actually a small subset of PL/I, at least volume-wise, and to include this information the language has been rechristened as MINIPL.

While one part, and the original motivation, of the project is to come up with the specifications for the aforesaid language, the other part is to experiment with its implementation. That the implementation was going to be experimental was realized in the very beginning, keeping in view the time constraint and the nature and size of the team of implementors-two M. Tech. students enturing afresh into the area.

The primary aim was to explore the idea of writing a compiler based upon a formal syntax directed technique and meeting certain other implementation criteria, viz., machine independence, partability, etc. The choice of the particular syntax directed technique to be chosen was influenced by the availability of a constructor for generating the tables that drive the syntax analyzer (23).

The present work is reported in the two theses subtitled Part I and Part II . The contents of the two parts reflect, to some extent, the division of work; the common stem of language design and major compiler organization decision is indicative of the joint effort in these stages.

In the introductory chapter we identify the user group and its needs, as also certain implementation requirements. Language design criteria in the context of the user needs, are developed in Chapter 2. In Chapter 3, we survey certain prominent languages, justify the choice of a subset of PL/I and come up with the specifications of MINIPL. Chapter 4 describes the major design problem and decisions, in addition to providing an over view of the present compiler. Chapter 5 tells about the experience of using the transition matrix technique.

Chapter 6 (Part I) describes the semantic analysis and also how the structure of a MINIPL program is checked. Chapter 6 (Part II) deals with the lexical analysis and describes the associated routines. In Chapter 7 (Part I) we discuss symbol table management, storage allocation as well as the run time addressing mechanism. Input/Output handling is dealt with in Chapter 7 (Part II) which also describes the generation of intermediate language output. In Chapter 8 we look back at the whole project in retrospect.

CHAPTER 1

INTRODUCTION

This chapter in addition to giving an introduction to the changing scene of language implementation and design serves to outline the aim and motivation of the present project. Section 1.1 devotes itself to the former while 1.2 brings out the latter by identifying a group of users and their needs. In section 1.3 we give a few important implementation requirements.

1.1. Languages and their Implementation : a brief look

There is a fundamental relationship between languages and the thinking habits of people. The language mirrors the thinking habits of those creating it, conversely, people are forced to think and to express themselves in the language. In particular, this is true for programming languages. Perhaps the influence of programming languages is even stronger, as the dialogue involves not only men but machines too, and the degree of precision and clarity required is more pronounced. The machines have, however, taken more than a fair share of the designers' attention - and that too not with respect to the criteria in the previous sentence. Early equipment was a painfully pinching shoe , and to push the machine to it's limits was thought to be all that was there to programming. This is reflected in the early high level

languages too where optimum utilization of the critical hardware resources screened the poor programmers interests. With the fantastic developments in the hardware technology the attitude should have changed early enough - but unfortunately it did not. This created a vast gap between the Hardware and Software capabilities and the computer world was faced with what J.D.Evans (11) termed as the 'Software Crisis'. In the past few years a lot of rethinking has been done as to what role should a programming language play as a tool in better programming practices.

The past decade has also seen a lot of activity in the area of definition and implementation of languages. The roots of systematization of the hitherto adhoc development in the area can be traced to the pioneering work done by Naur et. al (22) in the definition of Algol 60 syntax. The description tool used there for the first time (Backus Normal Form or BNF for short) is not only of use in precisely defining the syntactic features of the language, but has also been a great impetus to the host of new techniques for parsing and translation of languages based on the BNF description or it's variations. These techniques known as the syntax directed techniques, have come a long way from Floyd's paper (13) which described the relationship between syntax and programming languages and presented a top-down algorithm for analysis of arbitrary context-free languages. Since then, a lot of more efficient algorithms have been designed for restricted subsets of the context-free languages, which are important, however,

in as much as they cover most aspects of the present programming languages. Compared to the earlier adhoc techniques, these syntax directed methods not only make for more efficient processors, in most cases they provide a central theme to give a more integrated and intellectually manageable compiler. They make the extension and modification of the input language possible without major revision in the whole compiler.

The present project is the beginning of an effort on both the fronts - language design and implementation. The aim and motivation behind it can be more closely identified if we look at the needs of the would-be users and then at the additional requirements imposed upon the implementation.

1.2. The User Group and It's Needs :

The group singled out as the most likely users is that of would-be 'programmers', or to be more specific, the beginning computer (software) science graduates (or undergraduates as the case may be). The reason why this group has been singled out is that it is this group which hopefully will take up the task of bridging the hardware - software gap alluded to in the opening section. Also, it is hoped that because of the fundamental concern of this group with 'programming', it may bear or be made to bear with the discipline such a language might impose. The main purpose, the language is supposed to fulfil is as the main support language in a first course in programming for the beginning graduate student .

Now, that the identification of the users is over, let us take a look at their needs. It will be noticed that some of the needs were presupposed while identifying the group of users. This is, however, inevitable in view of the fundamental nature of the task involved, where what we call as needs have been burning issues of discussion for past few years. This is in contrast with the needs - analysis of developing a package for Civil Engineers or an inventory control system, where an established customer(s) lays out or helps in laying out needs which are more physical than the ones under consideration now.

To enumerate the needs, the language

1. should help develop a good 'Programming Style'.
2. should serve as a vehicle for the introduction of the student to the important techniques in programming.
3. should be easy to learn.

1.2.1. Programming Style :

The recognition of the first need has come from the change in attitude towards programming. The old belief that a gimmick-box is the best kind of programmer that can be, has been fast losing ground as the economic balance has tilted considerably away from the 'tricky' adhoc programming and towards the systematic, structured programming. (N. Wirth (36), E.W.Dykstra (7)). There has been a gradual shift of stress from making the programs efficient - though probably rendering it obscure in the process -

towards making programs that are more readable, easily amenable to abstraction - hence better mangable intellectually, and lending themselves to a correctness proof with some degree of hope. As the complexity of programs increases the intellectual managability and reliability become greater problems. The need is for the emergence of a style of programming which aids the decumentation, verification and systematic development of large programs. Although the language for the beginners may at times fail as the medium to implement huge sophisticated systems it can atleast guide the beginning programmer in developing such habits as will equip him better to meet the above mentioned tasks.

1.2.2. Teaching about Programming :

While programming style discussed above can, to a certain extent, be brought into the framework of a methodology, teaching programming techniques has to be tackled by exhibiting certain widely acceptable techniques. It is like a collection of tools, to master whose use, the underlying mental concepts can be taught only by teaching most of the important ideas that exist in the field. The task is rendered simpler if the support language provides some features for natural expression of some of these techniques - say recursion - data aggregates list processing and so on. However the list tends to get longer and longer and comes into conflict with implementational feasibility. Besides, too many ready-made advanced features might lead the beginner to what Dijkstra (5) called an 'addiction' of features. In any case the

language should be sophisticated enough to make it possible to illustrate the important ideas.

1.2.3. Ease of Learning :

A basic need a language must fulfil, especially meant for beginners, is that it should be easy to learn. Ofcourse, by beginners we do not mean students in high schools and general colleges for whom A.J. Perlis (24) predicts continued use of Fortran, as it is ~~very~~ easy to write simple programs in it. Our group of users is expected to be more motivated and ready to put up with the initial discipline. In this respect we shall consider, as a measure of the ease of learning, the total time required to get to write programs and write them well, learning about the whole thing in an integrated way, rather than taking up a tinkering approach. The baroqueness or the bulk etc. do come in the way and should be curtailed as much as possible.

1.3. Some Implementation Requirements :

The needs of the groups of users have been discussed in general. As to how they are to be met in the language, will be discussed in later chapters. The implementors also stand to benefit if the above needs are fulfilled in as much as it would have made for an experience with the ideas presented. The direct interest of this party, however, was to experiment with implementation of the language and the major body of the project has had to do with the implementation part. In addition to satisfying the above needs

there are some additional conditions which it is desirable to meet. One is that the implementation should be as much machine independent as possible -as it will help in wider availability of the language, once implemented. The implications of machine independence will be reflected in choice of languages etc. involved in the implementation process as also in the structure of the compiler. This will be discussed in detail in the chapters on implementation. Another requirement is that the organization of the compiler should be such as to permit intellectual managability and ease of extension and modification, both of the language as well as implementation.
The idea of the central compact organization implies the minimisation of phases in which implementation is to be carried out. This will result in less book-keeping and easier management of the job. A centralized organization is also important as it helps in easier control. These criteria of implementation will be taken up in detail in chapter 4. Leter chapters will describe various individual implementation tasks in detail as well as the difficulties involved.

1.4. Conclusions:

Motivation of providing a language to meet the needs of the beginning programmer has been established. Another motivating factor is the instructional value, to the implementors, of an experiment with compiler writing. In the next two chapters we shall chalk out in detail the language we want and go on to talk about implementational matters in later chapters.

CHAPTER 2

LANGUAGE DESIGN CONSIDERATIONS

The last chapter identified the group of people, to fulfil whose needs, it is desired to implement a new language. Certain needs were brought out and in this chapter we shall attempt to come up with the desirable and undesirable features-in so far as they contribute to or violate the fulfilment of these needs - for the proposed language.

A whole lot of opinions have been expressed as to the various features languages should or should not have. But a unified formal theory of language design does not exist. Most of the considerations are at best intuitive. The approach thus should be to survey and analyse the important ideas in the field with the hope that an outline emerges in the end.

It will simplify the analysis to compartmentalize our study into three sections - (1) Program structure and Control Flow (2) Data structures and their manipulation and the functional capabilities and (3) the factors affecting the ease of learning - corresponding to the three needs, programming style, teaching of important programming techniques and the ease in learning. It should, however, be noted that such a division can not be water-tight in as much as the factors covered under one head might well

contribute to the other. For example, the power to explicitly name and access subfields of a word, although an important contribution to the functional capability of a language, contributes significantly to program clarity and documentation. Similarly, ease of learning may be affected by the features contributing to the first two needs. This sometimes may pose an engineering trade-off and at others it may not - keeping in mind our definition of the ease of learning. Another engineering problem is that of desirability and practicability of providing certain features. This, to a certain extent, will be brought out in the present discussion but will be more completely discussed in later chapters when we shall specifically outline the omissions due to implementational difficulties.

2.1. Program Structure and Control Flow :

A program may be just a heap of statements or may have a definite structure reflecting a symmetric treatment of the subject. Programming style or its lack is manifested by the presence or absence of its various attributes - viz., readability, flexibility, reliability and modularity etc., although data structures and other functional apabilities do contribute to the programming style-the significant effect on the features listed above is of control flow and structuring facilities. In this section we shall discuss the issues pertaining to these two.

2.1.1. The GO TO Problem :

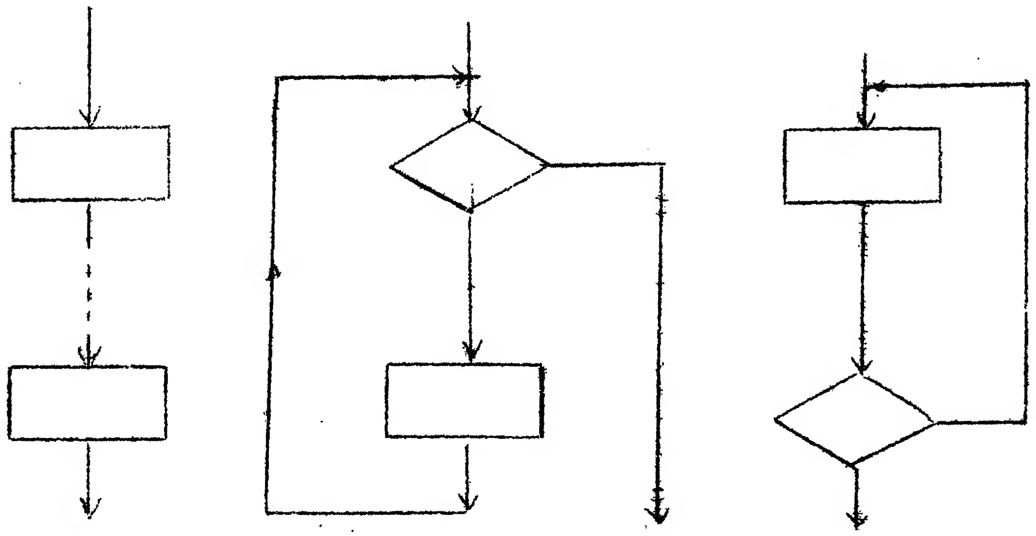
The only mechanisms for changing the sequential execution of statements in Fortran are the procedure, the do-loop and the unconditional and conditional jumps. Algol and other languages of its genre introduced certain other control mechanisms that occur frequently in most algorithms - notably the if-then-else, the while and repeat loops and the case statement. But most of these languages including the latest developed (Pascal (36)) have retained the free jump.

Recently there has been a great controversy - the issue was born with the letter from E.W. Dijkstra (8) in CACM - as to whether or not the free jump should be done away with. The argument against the go to was that it is possible to use go to in many ways which obscure the logical structure of the program, thus making it difficult to understand, debug and prove its correctness.

2.1.2. Dijkstra's Basic Constructs :

According to the Dijkstra school of thought four basic constructs are enough as the basic mechanisms in building a program. They are : concatenation, selection from a pair, the prechecking (while) and postchecking (repeat) loops, and the 'case' construct (Fig. 2.1)

Verification strategies for proving the correctness of these mechanisms using assertions regarding the states at the

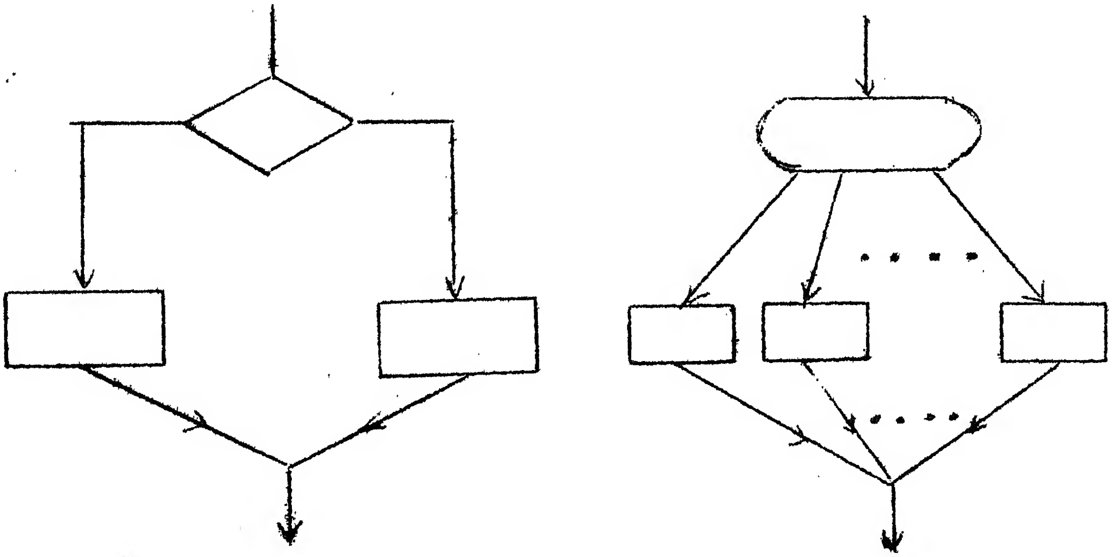


a) concatenation

loops

c) pre-checking

d) post-checking



b) selection from a pair

e) case

Figure 2.1

entry and exit to these basic blocks have been given by Dijkstra (7). Although the Dijkstra constructs (the constructs did exist from the beginning in Algol 60, but their role in program verification etc., is attributed to Dijkstra) do not provide any proof by themselves , they do lend themselves to a methodical step by step approach to proving the correctness. By definition (37), a go-to-less graph is susceptible to a sequence of transformations reducing it to a single process - box. Take a sequence where : (1) The correctness of the replaced construct has been verified, and (2) the new process-box contains a more macroscopic description of what the replaced portion does. This sequence forms both a proof as well as the documentation of the original programs. Taking the reverse approach to this bottom-up one, we can take a macro-box of the program description and gradually decompose it to finer and finer boxes till we reach the working program capable of interpretation by the computing environment. This approach to program design can better help in abstraction and control of program than while using free go-to's.

2.1.3. Extensions for Sake of Naturalness and Power :

Ashcroft and Manna (1) have shown how an arbitrary program with go-to's can be converted to the one without go-to. The results are however not of direct interest to us, as the introduction of a whole lot of state variables etc., seem to make the resulting program even more obscure. Thus the criteria should

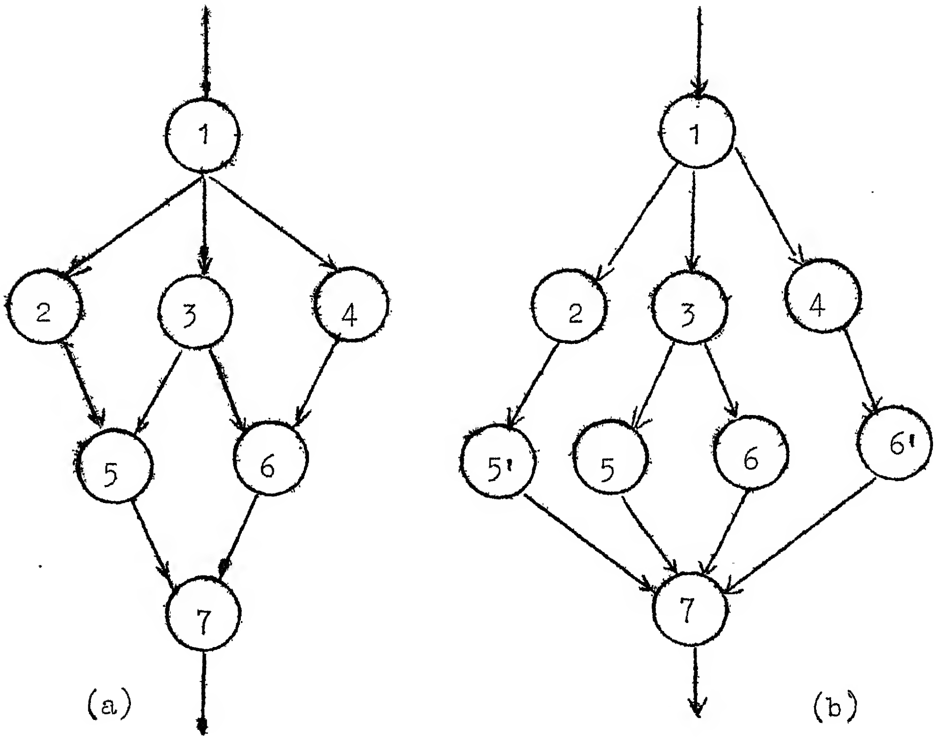


Fig. 2.2

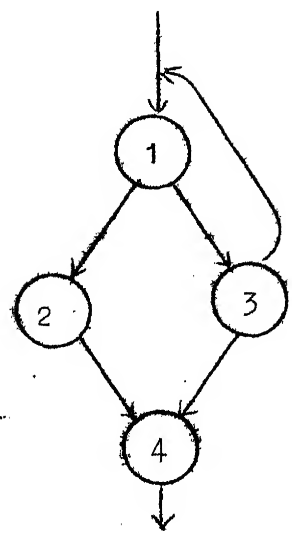
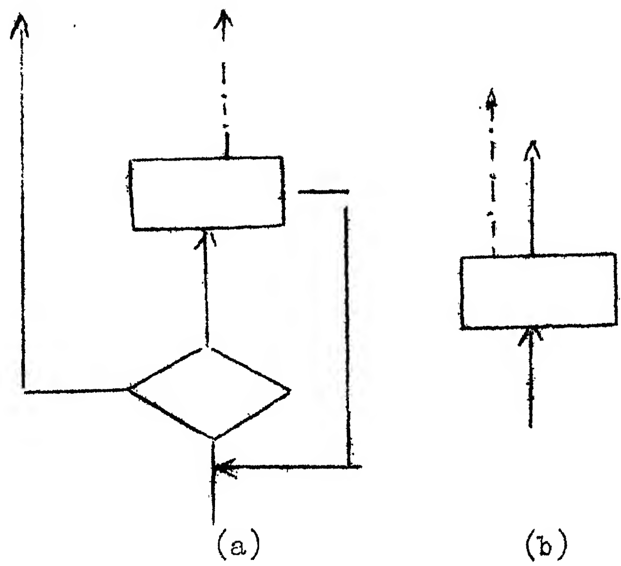


Fig.2.3



Dotted lines represent error exi

Fig. 2.4

be whether or not programs written afresh with the iterative application of the aforesaid constructs can express the entirety of structures in a natural way.

It has been shown by Wulf (37) and Peterson et. al.(25) that there are certain graphs, to transform which, node splittings are required (fig. 2.2a and fig. 2.2b illustrate this). This implies duplication of pieces of code. For certain other type of graphs (fig. 2.3 is an example) Wulf (37) has shown the impotency of the "node-splitting" technique. The use of Ashcroft and Manna type of state-variables is suggested as one way out but Wulf himself agrees that the technique is 'odious'. The latter problem can however be tackled by the use of multiple-level-exit loops. Wulf does it through his 'escape' construct allowing to escape from any of the surrounding control environments. Peterson (25) however proves that node splitting still remains essential. This anyway is not serious if the amount of code to be duplicated is small. If it is large, procedures can be used. It is here that internal procedures without parameter transmission can counter to some extent the inefficiency (in time) of this solution.

Higher level constructs are desirable not only for the sake of completeness but also to more naturally express some of the frequently occurring conceptual patterns. D.E. Knuth (21), while reviewing Dijkstra's "Noteson Structured Programming", pleads in his 'open letter' to the latter that 'not all go-to's are bad'. He has given an example of a situation where,

'one knows what he wants to do but has to translate it painstakingly into a notation that often is not well suited to the mental concept'. The action he wants is 'first do α , then β , then if γ we are done, otherwise do δ and we're in the same situation we started' - or in a concise notation :

$$\underline{\text{loop}}\ \alpha\ ;\ \beta\ ;\ \underline{\text{if}}\ \gamma\ \underline{\text{then}}\ \underline{\text{exit}}\ ;\ \delta\ \underline{\text{end}}\ \underline{\text{loop}}. \qquad (1)$$

The round about solution using the basic Dijkstra loops could be :

$$\alpha\ ;\ \beta\ ;\ \underline{\text{while}}\ \underline{\text{not}}\ \gamma\ \underline{\text{do}}\ \{\delta\ ;\ \alpha\ ;\ \beta\ ;\ \}\ ;$$

or δ^{-1} ; $\underline{\text{repeat}}\ \{\delta\ ;\ \alpha\ ;\ \beta\}\ \underline{\text{until}}\ \gamma$;

where δ^{-1} is some invented trick inverse of δ .

Surely both the solutions do look contrived and a natural linguistic construct for depicting (1) is desirable. But the 'escape' mechanism together with a do-forever type of loop indeed looks sufficient to take care of the situation.

Bochmann (3) has suggested a new construct - 'multiple exits from a loop without go-to'. The format is :

<simple loop>

<label> : <statement>

<label> : <statement>

⋮

<label> : <statement>

ended : <statement>

In case of normal termination ended statement is executed while for abnormal exits exit loop label can be used.

The construct meets the claims of being easier to optimize and prove correct than the one using free go-to. Its addition is of doubtful value compared to the increase in the baroqueness of the language, because the effect can be achieved without much loss of clarity with multiple level exits from compound statements.

One of the questions to be answered is whether the multiple level exits maintain the spirit of go-to-less programming. The transformations discussed earlier (page 12) are valid if dotted lines (fig. 2.4) are ignored. At some stage in the reduction process exit lines will be totally enclosed. Hence one can apply the former reasoning to subgraph from which no dotted lines emnate. After this attention must shift to this subgraph. This may or may not lead to a simpler form of graph, but in either case the process can be iterated. In this sense desirable properties of go-to-less graphs are retained. Also a proof for multiple exit loop has been given by **Clint et. al.** (20).

Concluding it seems that the benefit of the exclusion of go-to, in as much as it avoids the temptation to use it as an easy way out even when undesirable, seems to outweigh the inconvenience it may sometimes cause in very special situations which arise in very few practical programs. Peterson et. al. (25) have themselves conceded that trying to program without go-to has produced better programs and, more importantly, given better insight into the problem than when go-to was available.

2.1.4. The Stepping Loop :

Another control-flow construct that has come under critisism from E.W. Dijkstra (5) is the count-controlled-do or the stepping-loop. The idea of the control index which is stepped up every time chains the programmer so, according to Dijkstra, that he overlooks the obvious and simpler solutions in many cases. Probably that may be the case when it is the only high level iteration construct available. But in presence of the other more general loops it seems a useful tool in the many situations where the regularly incrementing counter is indeed central to the iteration. Also, being a special case of the repeat loop, it in no way violates the proof requirements.

2.1.5. Blocks, Procedures and Hierarchial Structuring :

"Procedure is one of the few fundamental tools in the art of programming whose mastery has a decisive influence on the quality of a programmer's work" (N. Wirth (36)).

Procedure is indeed the most important of established tools in a programmer's repertoire. It serves as a device to abbreviate the text and more importantly as a means to structure a program into logically coherent closed components. In the process of hierarchial development described earlier, the macro-steps in the top-down process can be replaced by procedure calls - the procedures being defined in the refined versions.

The use of procedures, however, seems warranted only when there is a repeated use of a 'sequence of statements' especially in a language where 'compound statements' are there to do the job of structuring. The blocks or the compound statements with declarations, in addition, solve the problem of locality of variables.

The above discussion has tacitly assumed internal procedures. They alone are like blocks with return mechanisms. External procedures though important for another reason described later, fail to provide the ease of transmission of variables naturally available to the internal ones in the surrounding environments. The internal procedures - and equivalently blocks with declarations - are like service routines taking the throughput variables from the environment and having some local work areas to accomplish the transformation.

2.1.6. Scope Rules :

The declarations in blocks help to serve two purposes. One, they provide a data structuring by clearly mentioning that certain data has relevence within the block only. Second, they help in freeing the temporary storage as soon as a particular block is exited.

However as far as structuring of data goes the Algol like scope rules suffer from one disadvantage (14) - hierarchial ordering requires that no layer uses the data of another layer. In Algol like structure this is guaranteed in one way only : No outer block can access data declared in the inner blocks.

Conversely, global variables are open to misuse. The equivalence of Fortran type of labelled common - partitioning the data in a natural way - is simply not available. A solution proposed by Goos (14) is to name the blocks and preclude specific blocks as being outside the scope (fig. 2.5).

```
begin; except  A  real  X ;          Scope of X
                  .
                  .
                  .
         level    A  begin ;
                  .
                  .
                  .
                     end ;
                  .
                  .
                  .
end ;
```

Figure 2.5

The storage allocation advantage of blocks depends mainly on dynamic allocation. While at it we must mention the importance of the static variables belonging to the particular procedures. Their desirability stems from the fact that at times the information from the past execution is important and static variables (or own in algol) are essential or atleast natural in many situations e.g. in statistics collecting routines.

2.1.7. Modularity and External Procedures :

One place where the external procedures seem to have an important advantage over the internal procedures is to serve better as program modules. By modules we mean units which can be described independently of other units and are capable

of being combined with others without requiring a knowledge of a particular module's construction. Such routines which are usually kept in program libraries - impose another requirement - that of independent compilation. Göos (14) has discussed the problem of independent compilation of blocks, but

complications arise in handling non-local references (not at the same level external are of course simpler to handle)). Dennis (4) argues about nonsuitability of internal procedures, because of the conflict arising, say, when two procedures referring to the same nonlocal y, but intending it to supply different information. Dennis goes on to suggest that the only linkage between the program modules should be through parameters. This removes the justification of internal procedures as modules completely.

External procedures are also essential as a simple means to incorperate machine code routines once the calling conventions etc., are properly observed.

2.2. Data Structures - Their Manipulation and Other Functional Capabilities

Since the purpose of a program is to accomplish some sort of computation on data, the characteristics of a language are profoundly affected by the data types and data structures that it provides and the operations that it allows upon them. New languages emerge now and then because the existing languages do not have suitable data representation or methods of operating on them which may be convenient to use in a new problem area.

For a language to have a reasonable power it is necessary to have at least fixed point, floating point (for numerical computations), character (for text processing) and bit (for boolean operations) data types. The flexibility and elegence with which they can be used depends on the operations available on the variety of data structures provided.

For illustration, consider Fortran - it is severely limited in the area of text processing because of the absence of character data type. On the other hand Snobol would be a suitable language for text processing because of it's string data types and the operations of matching, concatenation, substitution etc., of strings that it provides. Snobol is useless for numerical computations because it does not have integer and floating point data types and it's arithmetic expressions are cumbersome to form; though some small numerical computations can be done using string variables or constants which are composed of digits.

2.2.1. Structured Values :

A number of alternatives for defining new data types as a composition of the basic data types are possible. We call values with such a type structured values. A structure is a tree whose nodes are associated with names and whose end nodes have data values. Structures of different complexities are possible. Some languages allow structures having only two levels; others place no such restrictions and may thereby allow structures composed of minor structures which may themselves be composed of further substructures and so on.

Use of structured data type, where applicable, is undoubtedly an aid in increasing the readability of a program since it provides means of referring data by meaningful names. Besides, the description of the structure brings out clearly the relationship between it's data elements at one glance. Absence of structures would force the programmer to find this relationship by looking at the use of different data in program, which is a setback for easy readability.

Of all the languages PL/1 supports the most complex structures. Such complex structures find their applications in business data processing. Because of their complexity they are not suitable to be introduced to beginners. It is felt that structures like the Hoare records (16) which have only two levels - one node and any number of components - will be more appropriate since they can be easily understood because of their simplicity and they are powerful enough to be useful for most of the applications where structures are used.

Goos (14) has made a strong argument to allow to attach an identifier and a data type to a subfield of a computer word (by that he presumably means packing within the smallest unit, which may be a byte to, as character data stored in bytes are possible in some languages). In a way this facility is allowed when the implementation packs boolean data one to a bit. This however implies a uniform accessing mechanism determined by the compiler for data type boolean. As for securing a facility to pack differing data closely one can get it using structured values provided the range of components is specifiable. This, Goos (14) says, will eliminate the need for packing and unpacking

data which must be described by shifts etc. Thus, it will make for better readability by identifying the data by its type rather than by looking at the operation necessary to access the data. The facility seems to be available in some languages where ranges are specifiable but here too (14) there is no direct control by the programmer over the packing mechanism. In order to avoid generating different codes for accessing, the implementation may decide to allign the data on certain boundaries (e.g. word, halfword etc.). The Hoare records however are only accessible indirectly through pointers. This does seem to be an unnecessary limitation. The type of structure values given by Gries (16) for his example language, seems to be the best fitting to our needs.

2.2.2. List Processing :

In some problem areas programming is done more naturally if list processing capabilities are available. Need for list processing facilities arises when the data structure is elaborate and has to be taken into account and when this structure has to be operated upon as well as the values contained in it. Simulations of 'the more intelligent' types of computation, such as proving geometric theoremsin Euclidean manner, or cybernetic simulation, requires extensive use of lists. It is list processing techniques which have shown how to program a computer so that it can, for example, input a character string such as $ax + b = cx + d$ and manipulate it so that it can be output in the rearranged from

$x = (d-b) / (a-c)$. List processing is now included as an integral part of some new languages but it's utility in numerical computations is very little.

Although it is not our aim to perform sophisticated tasks of a specialized nature, the language should have facility for basic training, so that, later, sophisticated tasks can be taken up in a more sophisticated language specially suited to that task.

Methods of construction of list structures and manipulations upon them very from language to language. In PL/1 the use of pointer variables and address primitives alongwith variables, arrays and structures declared BASED permits the construction of arbitrarily complex address linked storage structures. The burden of linking and delinking storage and the allocation and release of storage is put on the programmer. Other list processing languages, like SLIP have primitives for adding or deleting cells at the end or the beginning of the list, or between two adjacent cells. Primitives exist for making lists as sublists of other lists and for traversing the list to examine the cells and manipulate the data in them.

Certain rules may exist to determine when a list can be returned to free storage. None of the forms for the components of list languages has as yet been established as 'standard' as it is difficult, at this stage, to specify the desirable components of a good list processing language. We must wait till these languages have evolved into a better state. Perhaps, it may be

established that it is best to have only the primitives essential for list processing so that the programmer gets full flexibility.

To this end, provision of pointer variables and means to access and store in them seem necessary. This takes away from the programmer's hands the explicit handling of links which is the case, say, when lists are simulated in arrays using indices as links. This, while reducing chances of errors, also makes for better clarity in programming. With structured values providing a means to specify different type of cells an adequate facility seems to be at hand. If, we get some means of borrowing and returning cells to the 'free space', we indeed have a powerful yet simple list processing primitive.

2.2.3. Recursion :

The wisdom of providing recursion in a language is often debated because it is possible to implement recursive computations in a non-recursive way; besides, recursion requires lots of memory space. In numerical work, often when the recursive definition is neater, the iterative process both looks and is more satisfactory in use. Some people look upon recursion as an expensive luxury.

We can think of recursion as having two main spheres of importance. The first is somewhat theoretical and is that recursive functions are the basis of the whole modern theory of computable functions. The second important use of recursion arises because the situation in numerical work is not altogether typical. In particular, procedures for analysing structures which

may be recursive are most efficient if they themselves are recursive, and they must at least incorporate features which would be unnecessary in the absence of recursion in the data. Tree structures, which are recursive, are very often encountered in numerical work. Since a tree consists of subtrees any operation on trees is most naturally seen as a cascading of identical operations on sequences of subtrees. Availability of recursion in such situations will allow the programmer to write elegant programs in a natural way instead of forcing him to twist his programs to do the recursive computations in a non-recursive way; an excellent example is the tree sorting (AVL trees) program given in (32). Another area where recursion fits in naturally is the evaluation of recursive functions. Ackerman's functions, for example is most easily defined recursively and it's evaluation by other than recursive means is extremely cumbersome.

Whether recursion should or should not be provided in a language depends very much on what needs the language is required to fulfil; but there is no reason why it should not be provided in a language like PL/1 or Algol which are based on dynamic storage principles and it's implementation would not require any additional features of significant complexity.

2.2.4. Input/Output :

No language is of any practical utility without some sort of I/O facilities. Indeed, ISO has made a provision of explicit input/output a prerequisite for recognition of a language.

Every programmer, including the beginner, has to atleast output his results; it is pointless to even write a program which is like a dumb giant, even worse than a mumbling idiot who takes no input and only outputs. The analogy is taken from Beizer (2) and transfixed upon program instead of computer. Therefore a beginner must learn I/O instructions before he has fully grasped the basic concepts of the language. Keeping this in view the set of I/O instructions must include some very simple instructions which will enable the beginner to input/output his data/results without bothering too much about the layout. The formatless I/O statement has reason for existence primarily to cater to the needs of the beginner. It's popularity in the undergraduate programming courses is a known fact. Ofcourse, formatted I/O facilities should be available also, for more mature programmers to have control over the layout of the graphics.

Among the higher level languages I/O may be consist of two broad classes : stream oriented and record oriented. Stream - oriented input deals with a continuous stream of characters. On input, data items are selected one by one from the stream of characters that are converted to internal form and assigned to variables specified in a list. Similarly, on output, data items are converted one by one to external character form and are added to a conceptually continuous stream of characters. Record - oriented input - output deals with collections of data, called records, and transmits these a record at a time. Stream oriented input/output is conceptually easier to understand because it avoids

the concept of records and follows the natural line of thinking that the next item to be input/output follows the previous one unless an explicit control is exerted to change this layout.

Following the general principle of naturalness, a language should be able to handle the I/O activity in a natural way. The most natural way to specify when and under what conditions an I/O operation should take place is simply to command it to take place at the right time. The roundabout and inelegant mechanism to handle I/O in languages not having commands is worth noticing. In Lisp 1.5, 'pseudo functions' are evaluated for their side effects, such as a print operation, and their true values are ignored; in Snobol, data can be output only by the awkward mechanism of requiring it to be a part of the special string named SYSPOT. The language should be able to handle a list of variables for I/O in a natural way without having to resort to list procedures (as in Algol), without any restrictions on the number of parameters and without having to make use of structure names, array - names, or names of pushdown lists artificially created for the purposes of I/O.

2.2.5. Operations On Data Aggregates :

The influence exerted by the set of operations present in a language on the programming can not be ignored. The sophisticated notations and operations of APL make the programs very elegent and concise. APL directs a programmer to organise his activities on arrays of data. A large set of operators are provided for manipulating these arrays. The variety of available operations

permits the programmer to express an amazingly large cross-section of useful algorithms in a concise and natural way (24).

The language owes it's advantages and it's difficulties to it's heavy use of operators. It's large set of operators is a heavy burden on the programmer. While an experienced, professional programmer may love to use it for it's natural and concise programming, it's notational complexity makes it an impractical language for a beginner. Here, elegance of programs and ease of learning are in direct conflict.

In PL/1, one can perform operations on structures, arrays and subarrays. While it may be claimed that it is a convenient feature which allows for succinct programs, it can not be denied that it is more a luxury than a necessity. How far can we go in providing such luxuries ? The beginner should better get down to do these operations element by element and learn the basic fundamentals of programming.

Structure and array operations do not stand condemned completely; they do have their utility in special purpose tasks. It is fully justified to have structure operations in Cobol but it certainly has no place in a beginner's language.

2.3. Factors in Ease of Learning :

We have already discussed some of the factors which affect programming style in the sections dealing with block structures, data structures and functional capabilities. Apart from the elegance they

lend to the program, some of the features, such as recursion, list processing and structured values have their utility in a beginners' language since they introduce some of the important concepts in programming.

In this section we look at the factors affecting ease of learning. One of the major obstacles to ease of learning is too large a size of the language. Some of the modern languages, representatives being the two 'Omnibus' languages PL/I and Algol 68, are so monstrous in size as to frighten away the user completely. Of course size in direct proportion varies with facilities provided but it is ultimately a question of architecture where the extra features cease to contribute to a functional and simple design and start transforming the languages into what Dijkstra calls 'baroque monstrosities'. While agreeing that volume should be kept within managable limits - both mentally and mechanically, let us look at some of the other factors affecting ease of learning.

2.3.1 <u>Uniformity</u> :

This term means that the same thing should be done in the same way whenever it occurs and that the same syntactic construction should not mean different things in different things in different contexts. Some identifiers in PL/I, for example, will be treated as special words or ordinary identifiers depending on the contexts. It should be clear, without any elaboration, that uniformity is an aid towards easy learning. An example of

where it does not happen is the implicit declarations of Fortran which have been carried over to PL/I. If one intends to use PL/I as FORTRAN (yes, it can be done !) then its fine. But once we tell the beginner about the block structure and so on, he will find by sad experience that implicit declarations (a habit which he is not required to relinquish) which he thought he had made for an internal block, already span the whole external procedure-scopewise.

2.3.2 Restrictions:

Many languages, notably Fortran among them, impose irrational restrictions on the use of some of the constructs. Fortran allows very restricted use of expressions as array indices. The result of this restriction, a survey has shown (34), is that many users avoid using any sort of expression in the array indices because they find it difficult to remember the restrictions. Another example, again from Fortran, is that of the DO-loop- index. Severe restrictions are put on it by not allowing the index to be initialized by an expression, or its final value to be specified by an expression. Not only the step size cannot be specified by an expression, downward counting is simply not possible.

The restrictions were placed to make the compiler design easier. But keeping in mind that the language is designed in the first place for the programmer rather than the compiler writer, no new language should have no such restrictions.

2.3.3. Explicit Declarations:

Explicit declarations, especially when one is forced to put them in the beginning, provide for good documentation by reminding one to describe them all at one place. Use of explicit declarations aids in detecting extrenuous data names possibly introduced by mis-spelling. The following example will illustrate the point.

LBAT = LTAB + 1

The intention obviously was to increment the counter by one but due to mis-spelling Fortran allocated two locations for what was supposed to be one variable. The bug might take any length of time to detect. By requiring explicit declarations of variables, such errors will be detected by the compiler.

Similarly providing all the procedure definitions in a block at the beginning also makes for better comprehension.

2.3.4. Error Prone Features:

Some languages have features which are difficult to master and even when these features are properly understood there is a high probability of making mistakes when using them. An example at hand is the call by name feature of Algol. The concept is difficult to be understood by a beginner, and when making use of it he can easily get himself entangled in trying to keep track of the different values that the formal parameter whose correspondence with the actual parameter is by name, may assume during the execution

of the procedure . PL/I, which is based on Algol, has done well to exclude this feature.

In general a language designer should shun any such feature the use of which is heavily error prone and therefore whose exclusion is more beneficial to the programmer than the inclusion. In light of the discussion of 'tricky' V/S 'methodical programming' (5) has questioned the prominence given to the so called Jenkin's Device in many introductory books to Algol.

The other example has to do with the syntax of a very improtant structured programming construct-the 'CASE'.

Take the Algol W <u>case</u>:

```
Case I of
 begin
     A;
     B;
     C;
 end;
```

Frequent errors arise from the case construction due to the omission or rearrangement of cases.

Steele and Sedgewick (31) have incorporated a nicer case which goes as follows:

```
        CASE IVAR
       2 : .
            .
            .
            END
       4 : .
            .
            .
            END
       5 : .
            .
            .
            END
    ELSE : .
            .
            .
            END
```

In this the case is selected depending upon the value of IVAR but the possibility of the above type of error is now almost eliminated but for the presence of ELSE.

2.3.5. Error Messages :

This, strictly speaking, is not a feature of a language but of its implementation, but might be discussed while we are at it. Good compile time error messages do go a long way to help the learner in the initial stages to learn about the correct structuring of programs and proper use of the other syntactic features

fast. Of course, extensive error checking also helps the novice in an indirect fashion by saving him from getting bogged down in solving the puzzle of where exactly the program went wrong. Run time error messages fall in this category. These, however, are best discussed in the light of implementation.

2.4. Conclusions:

In this chapter we outlined some of the important considerations and principles in language design, but since a unified formal theory of language design does not exist, the elucidation of programming design principles is more of an empirical matter. Personal tastes and preferences vary. It is hoped that what has been recommended suits the needs of a large majority of programmers.

In the next chapter we come up with the specifications for a language, based on the principles discussed in this section.

CHAPTER 3

SPECIFYING THE MEDIUM-OF-INSTRUCTION LANGUAGE.

In the last chapter a discussion was given on the desirable and undesirable features in a programming language meant to serve as a medium of instruction for the beginning programmer. Based on the broad guiding principles which evolved from the discussion we shall attempt to come up with a language for use in an introductory programming course, either by designing a new language or, if possible, by suitably modifying an existing language to fit the requirements. The latter approach has several advantages over the former.

Firstly, it is less time consuming; developing a new language from scratch is a tremendous task and time limitations put it beyond the scope of this project. Secondly as there are already too many languages and, to make it worse, they are proliferating at a rapid rate; it is not advisable to increase the existing confusion by making an addition to the Babel of languages. Another point against designing a new language is that of acceptability, To make a new language acceptable it is an advantage to have proper commercial and political backing without which it just flounders away. If a modification of a commercial language becomes possible we stand to gain a part of this advantage . Another point to considered is that people

are reluctant to learn a language, however good it may be, if it is not widely available. Obviously a new language will not be widely available to start with, and it cannot be made widely available without the vital commercial push. If a suitable language to provide the general format becomes possible, the compatibility gained is a definite advantage and goes part of the way to meet the need mentioned in the previous sentence.

3.1. A Survey of Some Prominent Languages:

In order to choose a language on which to base the design of our language, a survey must be made of the existing languages, keeping in view of our requirements of a language for novice programmers. There is no need to consider all the languages because many are for special purposes such as those for string processing, list processing, simulation, for civil engineering, logic design, compiler writing and so on. Many others have died a natural death because of their uselessness.

We are interested in a language suited for a general purpose work for the intended community of users. On the forefront, in this category of users are Fortran, Algol and PL/I. We shall compare these languages and choose the best and see whether it suits our needs. No attempt is made to make a detailed comparison; only those outstanding advantages or disadvantages are considered which help us to conclusively discard one language or accept another.

3.1.1 Fortran :

It is probably safe to say that, today, more computer programs are written in Fortran than in any other programming language. Because of this the use of Fortran is well entrenched. Programmers working in Fortran are many, and their work forms voluminous parts of the program libraries. Its current entrenchment tends to ensure its continuing use even though this continuation is otherwise logically or economically indefensible. A.J. Perlis (24) while describing and condoning the widespread use of Fortran accepts that probably Fortran is more like a weed than a flower - it is hardy, occasionally blooms and grows in every computer.

Among its plus points are that it is a simple language, versatile and probably has the most efficient compile time and execution-time implementation on a majority of systems. The original design of the language included several IBM 704 dependent constructions which remain as some of the weaker features of the language at the present time. For example the Fortran iteration statement is subject to constraints based on the index register characteristics of the 704. It does not allow the use of expression in count controlled statement. It's other inadequacies are that it has under-developed data types and structures and restrictions on subscripts and expressions.

The most serious drawbacks, however are its lack of character data types, poor decision facilities and, above all, it's poor

program structure. Character data is an important thing for the training in text-processing. Of course it can be done in Fortran but in a round about and machine dependent way. The poor program structure makes it unsuitable for structured programming. Since we consider structured programming as the essence of good programming style this drawback alone forces us to reject this language. Even if we grant that it is a simple language and easy to start off for the beginner, it is definitely not suitable for expressing well and is consequently harmful in the long run.

3.1.2. <u>Algol</u> :

The appearance of ALGOL marked a very great step forward in the subject of programming languages. Of the various technological contributions that it has made in the field, the ones of major interest to us, are (1) a general simplicity combined with power for stating computational process, (2) block structure and defining the scope of variables, and (3) recursive procedures.

The first point is a general one, namely that Algol is a clean language of great power for expressing algorithms to solve a wide class of problems. The basic purpose of Algol was to 'describle computational processes' and it was achieved by making it a problem statement language.

Its block structure allows unlimited levels of nesting. The block concept serves the the purposes of combining statements into groups for control purposes, of indicating the scope and range

of definitions for the names locally, and of defining a procedure which can be called from different places in the program. This feature makes Algol one of the languages suitable for **structured programming**.

As we discussed in Chapter 2, recursion is one of the important programming concepts which we wish to introduce to the beginner, and this comes natural in Algol.

This discussion would imply that ALGOL would be a suitable language for our purpose since it is a clean language which encourages structured programming.and also has facilities for recursion. It certainly has a great edge over Fortran. It is much freer of restrictions and exceptions than Fortran. But as we shall see in the next section, PL/I is more suited for our purpose because apart from having these plus features of Algol it has much more.

Input/output, one of the essential features of all programming languages is poorly developed in Algol. One has to go out of one's way to perform these functions. Recalling that input/output is one of the areas where beginners find greatest difficulty and that this is one of the first things he must learn;it is a great drawback of Algol as far as a beginners language is concerned.

3.1.3: PL/I :

PL/I is a development triggered by what people often do and could not easily do with Fortran, Algol and Cobol. It combines all possible worthwhile features from algebraic, data processing, and

control languages into one unified polyglot. PL/I offers a programmer a great amount of power and flexibility. It supports almost all the features of Algol (except call by name); it has most of the data structure features of Cobol without its cumbersome syntax and annoying restrictions - in short, it is a very general language with the widest scope.

One of the objectives of it's design was to take a simple approach which would permit a natural description of programs so that few errors could be introduced during the transcription from the problem formulation into PL/I. This simplicity arises from the uniform rule that each statement be terminated by a semicolon, thereby removing the confusion of the sort present in Algol where the <u>end</u> delimiter sometimes has a semicolon following it and sometimes it does not. This simplicity at the statement level has been offset by it's complexity due to it's large size.

PL/I is modelled on Algol, accepting all it's good features and rejecting it's error prone features (e.g. call by name and the switch discussed in Chapter 2). It offers all the advantages of Algol, is much simpler and contains some additional programming concepts. PL/I, therefore, has an edge over Algol as a language for the beginner if only the major drawback of its mammoth size and baroqueness could be taken care of.

3.2. <u>PL/I vs Algol</u> :

We will now compare Algol and PL/I more specifically to highlight the relative advantages and disadvantages of the two.

PL/I has a more developed program structure than Algol. In Algol procedure ... end is used to delimit a procedure which may then be called from several different places with different arguments while begin ... end are used to delimit the scope of names to group a set of statements for control purposes and to specify the duration of allocation of storage for variables. In PL/I, however four syntactically different methods are used to accomplish the four function. In Algol by specifying a block to delimit scope of names, but not to be called out of line, it seems inefficient to be prepared to store register contents and return locations. PL/I avoids this by using PROCEDURE ... END for delimiting procedures and delineate the scope of names. BEGIN... END is used for delineating the scope of names. It may also be used to group a set of statements for control purposes. The grouping of a set of statements for control purposes seems to be a much simpler and common structure than one in which the scope of names is delimited. Further, it seems confusing to define a sequence of statements delimited by begin and end as syntactically different due to the accidental declaration of a single local variable. Therefore in PL/I DO and END are provided for grouping purposes. As far as specification of the duration of allocation of storage is concerned the artificial correspondence between the scope of names and storage allocation in Algol is rejected by PL/I in favour of allowing the programmer to specify the appropriate attribute for storage allocation.

PL/I has a much broader and flexible I/O facility than Algol. Unlike Algol in which I/O is done in a round about way I/O in PL/I is very natural and simple. Of special interest to us is it's simple stream oriented format free I/O particularly suitable for the novice.

Keeping in mind the difficulties caused by the call by name feature of Algol the designers of PL/I have studiously restricted it. Apart from this call by name feature PL/I encompasses almost all the Algol features. In PL/I only simple variables are called by name.

The data aggregates of PL/I are much more developed than those of Algol. The concepts of data structures and cross-section of arrays are complety missing in Algol.

The list-processing capabilities provided by PL/I by the use of it's pointer variable, controlled storage allocation and data structures are also absent in Algol.

Although it has no significance for a beginner's language, for the sake of completeness we mention that PL/I has several advanced features such as multi-tasking, compile time facilities and control over storage allocation, which are not found in Algol.

The introduction of such a variety of facilities in PL/I has not been achieved without paying a price for it - it has become a language of gigantic proportions ; the size of it's manual alone intimidates the beginner. Although one would desire that a user could ignore some of it's syntactic and semantic details

unrelated to his application, experience (24) has shown that the side effects which can arise do not permit this very often. It's otherwise free-from-error-prone usage is marred by it's complexity due to size. Many people have criticised it for it's bulk. To quote two:

1. 'The language itself as well as it's description have assumed remarkable dimensions and PL/I seems to be ill suited as a basic introduction to programming because of it's sheer size and it's lack of a systematic structure with a unifying underlying conception' - N. Wirth (36).
2. '... PL/I, a programming language for which the defining document is of a frightening size and complexity. Using PL/I must be like flying a plane with 7000 buttons, switches and handles to manipulate in the cockpit ' - E.W. Dijkstra(5).

3.3. The Required Language- a Subset of PL/I :

It is clear that PL/I, in it's totality, is a large complex language unsuitable as a beginner's language, although it's basic constructs and program structure are simple for comprehension by a beginner. Therefore, to make it acceptable as a beginner's language its size must be cut down drastically, retaining all the desirable features and none which is not going to be used by the beginner and whose presence may cause unnecessary difficulties for him.

A subset of PL/I chosen on these lines will have numerous advantages over the full PL/I. Namely :

1. The programmer will not have to remember those features which he is not going to use but the ignorance of which may give erroneous results.
2. Full PL/I requires a large system for implementation. A subset of it can be easily implemented on smaller machines and made available in all computer centres instead of only in the large ones. If minicomputers come in vogue, as is being predicted, PL/I may soon become obsolate; only it's smaller versions may survive.
3. A smaller subset is easier to implement and has more efficient compilation.

In sifting the subset out of PL/I we must have some guide lines for it. The guidelines are very general and not very technical in nature; they are set more by commonsense then by any formal principle.

In chosing the subset we must obviously include the basic essential features such as the assignment statement, expressions etc. without which it would be meaningless.

It must include those constructs which help meeting the needs of the intended community of users. These have been discussed in Chapter 2 where some empirical considerations in language design were given.

Some features, such as the compile time computation, multi-tasking, the more complex I/O mechanisms etc., which are irrelevant to our goals are to be rejected outright. Those features, which may although be

desirable, but have only marginal utility may be excluded to restrict the compiler to a manageable size.

Certain error-prone features, like mixed expressions, should be excluded.

In choosing the subset care must be taken to keep it a pure subset of PL/I. Purity is desired so that the beginner can switch to PL/I painlessly and so that programs written in the subset can be run on a machine where PL/I is available.

The title of the thesis only supplied four letters for the name of our language MEDIUM-OF-INSTRUCTION PROGRAMMING LANGUAGE. Now the justification for inclusion of the other two letters N and I making it MINIPL - the language is really a minature PL/I, at least in volume if not in substance.

3.4. The Language MINIPL - Specifications :

Describing the MINIPL fully is a big task and will acquire the proportions of a moderate sized manual. A way out of this problem is to describe it by comparison with PL/I, pointing out those features of the parent language which have been excluded and sometimes those which have been included, depending on which one is more economical spacewise. Of course this method of description assumes a fairly good introduction to PL/I. A quick-check syntax diagram is however provided in Appendix A.1 which will help settle disputes, if one arises.

The description will also incorporate the reasons for inclusion (or exclusion) of the features wherever necessary.

The semantics associated with MINIPL constructs are the same as those for PL/I except where the restrictions have been explicitly mentioned.

Not all the features of MINIPL have been implemented. The extent of implementation will be brought out in later chapters. What we describe here is the set of features which it is desirable to implement.

The following features of PL/I which are advanced and not necessary for the beginner have been excluded in accordance with the guidelines set earlier.

1. Multi-tasking and all it's associated data types and operations.
2. All compile time facilities.
3. Exception condition handling and program check-out.
4. Generic names and references.
5. Editing and string handling.
6. Record oriented transmission.
7. Passing arguments to the main procedure.

The following features are excluded to keep the language manageably small for implementation. Their utility for a beginner cannot be entirely ruled out but since our main goal is to provide facilities to encourage good programming habits and to teach the important programming concepts, and not, to give

each and every luxury, these features can be left out.

1. Function procedures.
2. Array expressions and structure operation. It is felt that the beginner will get a better feel for programming if he does these things by himself, element by element.
3. Dynamic arrays - they require a more complex storage allocation and therefore have been excluded.
4. The general mixed expressions of PL/I are not allowed because they are very prone to errors and are doubtful in aid to program clarity. It is described in greater detail when expressions are explained.
5. The use of GO TO statement has been restricted to improve the program structure. It can be used only for exiting from the control environment. Free jumps all over the place are prohibited to maintain the neat structure of GO-TO-less programming.

In the following pages we discuss the major features of MINIPL which include the program structure, data types and data structures, statements, expressions and program elements.

3.4.1. <u>Program Structure</u> :

Statements of MINIPL can be organised into blocks to form a program. Control may be passed within a program from one block of statements to another.

The structuring of a program into blocks serves the purposes of

1. Delimiting or defining a procedure which could be called from different places in a program with different arguments.
2. Indicating the scope and range of definitions for the names locally.
3. Combining of statements into groups for control purposes.

The structuring is done by procedure blocks, begin blocks and do groups. The ability to structure a program in this fashion gives a major support for doing structured programming.

The rules for forming the blocks using the program structure statements (i.e. PROCEDURE statement, BEGIN statement, END statement and ALLOCATE and FREE statements) are the same as in PL/I. The exception being that the END statement may not be followed by a label constant -the documentation can be done by putting /* label */. The declare statements must come in the beginning of a block just after the internal procedures. This rule is imposed for better program layout and programming discipline. Having data descriptions in the beginning improves the readability of the program. It also makes implementation easier and more efficient because it ensures that all the attributes of the variable are available to the compiler before usage. From similar considerations of implementation ease, program layout and programming discipline the internal procedures have to come right in the beginning of procedure block in which it is nested.

Some examples of blocks :

Procedure block

```
A .. PROCEDURE (X,Y,Z),.
     B.. PROCEDURE.,.
          :
          END /* B */,.
     DECLARE   X FIXED, ...
               .
               .
               .
     END  /* A */,.
```

Begin block

```
BEGIN ,.
     DECLARE Statement
       .
       .
       .
END /*  CDX  */,.
```

<u>Begin Block Termination</u> : A BEGIN block is terminated when any of the following occurs.

(1) Control reaches the END statement of the block.

(2) The execution of a GO TO statement within the block transfers control to the END statement for the current block or for the block in which it is nested.

(3) A STOP statement is executed.

(4) Control reaches a RETURN statement that transfers control out of the begin block and out of it's containing procedure as well.

Procedure Termination :

A procedure is terminated when one of the following occurs:

(1) Control reaches a RETURN statement within the procedure. The execution of a RETURN statement causes control to be returned to the point of invocation in the invoking procedure.

(2) Control reaches the END statement of the procedure. It may happen during the sequential execution of the program or by the use of a GO TO statement.

(3) The execution of a GO TO statement within the procedure (or any block activated from that procedure) transfers control to a point not contained within the procedure. If the transfer point is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are terminated.

Example:

```
      :
      CALL  A ,.
      :
      A.. PROCEDURE ,.
          B.. PROCEDURE,.
              :
              GO TO AE,.
              :
              END /* B */,.
```

```
      C.. PROCEDURE,.
           :
           CALL B,.
           :
           END /* C */,.
           :
      CALL C ,.
   AE.. END /* A */,.
```

In this example assume that procedure A is active and it activates procedure C, C activates B. In B, the exeution of the statement GO TO AE terminates procedures B,C and A.

Storage Allocation : There are three storage classes - static, automatic and based. Static allocation is done if it is desired that the value of the variable be preserved upon exit from a block. All variables that have the STATIC attribute are allocated storage before the execution of the program begins.

A variable that has the AUTOMATIC attribute is allocated storage upon activation of the block in which that variable is declared; it is freed when the block is freed. There is no provision for an explicit declaration to give AUTOMATIC attribute. Any variable which has not been explicitly declared as STATIC or BASED is given the AUTOMATIC attribute by default, with the exception that any variable that has the EXTERNAL attribute is assumed to have the STATIC attribute.

The BASED storage class has been provided for list-processing. The details of its use are to be found in section 3.4.4.

Recursion : An active procedure can be reactivated from within itself or from within another active procedure. Such a procedure, called recursive procedure, must have the RECUR option specified in it's procedure statement. The effect achieved by recursion in MINIPL is the same as in PL/I.

The facility for recursion has been provided to give the beginner an introduction to programming concepts in accordance with the decision taken in Chapter 2.

Subroutines: A subroutine is a procedure that is invoked by a CALL statement and usually requires arguments to be passed to it. When a subroutine is invoked, a relationship is established between the formal and actual parameters; the type of the two must match otherwise it would give erroneous results.

Functions and System functions : The functions have not been provided because they are difficult to implement; the system functions, however, can be incorporated when the need arises. System functions can be regarded to be operators belonging to a class of their own.

3.4.2 Data Elements:

The data types in MINIPL may be divided into two categories- problem data and program control data. The problem data is used to represent values to be used in processing. It consists of arithmetic, character string or boolean data types. The arithmetic data has a scale- either fixed or float. In MINIPL a variable

declared to be of fixed-point scale is taken to be of arithmetic data type of binary base and a floating-point declaration gives it the decimal base. The arithmetic data constants are all decimal. The fixed-point case is similar to Fortran where the arithmetic data constants are decimal but have a binary internal representation.

The programmer is not given any control over the specification of the precision in the declare statement. The precision is equivalent to the default option of PL/I and is set by the compiler and depends on the implementation.

The character string data has been restricted to length one for all purposes except when used in a PUT statement to output a string constant. This exception has been made to give the user facility somewhat equivalent to the outputting of the hollerith string in the FORTRAN format statement. For all other purposes the character string data can be considered to be character data since it has unit length. The character data has been provided for text processing.

The bit data type is the PL/I bit string of unit length. It has been provided for boolean computations.

An example of declaring the data types of variables is given below:

```
DECLARE  A FIXED,  B  FLOAT, C  CHAR, D  BIT, E POINTER;
```

The POINTER variable is used to point to a location. The value of a pointer is an address of a location in storage. The pointer data is included to do list processing.

3.4.3. Data Aggregates:

In MINIPL the important data aggregates have been included, namely: arrays and structures although they are not of the same complexity as in PL/I.

Arrays : There are no dynamic arrays in MINIPL. The reason for their exclusion has already appeared in 3.4. The value of the lower and upper bounds of the dimensions may be selected by the programmer. For the sake of uniformity both the bounds must be specified in a DECLARE statement instead of taking the value of the lower bound to be one by default as is usually done in most of the languages.

An example of declaration of an array:

```
DECLARE   X(-20.. 40)  FIXED ,.
```

Data Structures: The basic facility to define structures values as outlined in chapter 2 corresponds well with the PL/I if we limit the levels of structure to 2. Thus at level 1 we give the name of the structured value and level 2 the components. A typical declaration will be.

```
DECLARE   1  STRUC (5),
             2  LETTER (10) CHAR,
             2  AGE FIXED,
             2  PONT   POINTER,.
```

This declares an array of length 5 called STRUC - each of whose components is a structured value of the type

If means to declare ranges of numbers etc. were available (to a certain extent it is possible in PL/I but has been excluded because of implementational complexity), then we shall have the ability to access subfields of a word (by that, we mean fields that we get when packing chars in floating point and fixed point number cells). PL/I does provide varying ranges for integers but even PL/I(F) which is a fairly big compiler puts integers in only two forms half ward and full word. MINIPL does not include range specification facilities, presently.

Reference to a component of structured value is to be explicitly qualified in MINIPL, that is, structured variable take the following forms: A.B. , Q(5). P , S(5). T(3) , U.(5). Subscripts must immediately follow the identifier, unlike PL/I, where, S. T(5) is permissible.

Pointer Variables : We discussed in Chapter 2 the necessity to provide pointer variables. Gries (16) gives two functions for pointer variables meaning of which is clear from the example below.

P = ADDRESS (A) (1)

B = CONTENTS(P) (2)

CONTENTS(P) = B (3)

where, A,B are say fixed point variables and P,Q pointer variables.

(1) sets the pointer to point to A.

(2) stores the contents of the cell pointed to by P in B.

Thus the effect of (1) and (2) is the same as B = A.

PL/I declaration for pointer is,

DECLARE (P,Q) POINTER ,.

PL/I provides exact equivalent of (1) which is ADDR. As for B there is no direct equivalent and one has to use the based variables. Now we have to declare,

DECLARE ACCESSOR FIXED BASED (XXXXX),.

where in normal PL/I XXXXX is a pointer associated with BASED variable ACCESSOR. We shall not have occasion to use it as we shall only be using ACCESSOR to access values pointed to by other pointers say P and Q. The equivalent of CONTENTS(P) will be,

P ACCESSOR

or in MINIPL P PT ACCESSOR

3.4.4. <u>List Processing</u> :

Structure variables with pointers as components give a very powerful means of providing list processing primitives. The facility of Gries' language (16) when CONTENTS (P) becomes synonimous to structure name B if P is pointing to B is simply not possible.

The equivalent of CONTENTS (P). will have to make use of a based structure. It will be done as follows :

```
DECLARE  1 ACC    BASED (XXXX)
         2 R ...
         2 S ...
         2 T ...

P   PT   ACC . S
```

NULL : In addition to addresses a pointer variable may be assigned NULL, which means a quantity which is no address at all.

Pointers can be tested for equality among themselves as well as against NULL.

The major advantage over the SLIP like languages is the possibility to have cells with different sizes and layouts. Gries (16) has not proposed any methods of getting or returning cells, a facility of crucial importance if full benefits of list processing are to accrue. The facility becomesvery simply possible if we can provide the based storage. The suitable formats seem to be

```
ALLOCATE    A ;
```

where A is a based structure declared as

```
DECLARE  1  A  BASED (XXX),.
            2  ...
            2  ...
            2  ...
```

The address of newly allocated cell will be put in XXX . This cell can now be linked in the list being used by assigning the value of XXX to some pointer variable in list. Similarly using

FREE PONT PT A,.

will free the cell being pointed to by PONT the cell size will be determined by the size of A. That is, if one ALLOCATES A and FREES B (same pointer qualifier) it is his own funeral.

The features may be included in MINIPL if the special storage allocation (discussed in chapter no. 7 (Part I)) becomes feasible. The format for freeing and allocating will be as discussed above.

3.4.5. Statements :

A MINIPL program is constructed from basic program elements called statements. There are two types of statements - simple and compound. These statements make up larger program elements called groups and blocks.

A simple statement can be a null statement.

e.g. of simple statements

A = B + C ,.

GO TO START,.

A compound statement is a statement that contains one or more other statements as a part of it's statement body. There

is only one compound statement- the IF statement. The final statement of a compound statement is a simple statement that is terminated by a semicolon. Hence, the compound statement is terminated by this semicolon.

e.g. IF A GT B THEN A = B + C ,.
ELSE GO TO R,.

Unlike PL/I only the END statement may be labelled in MINIPL, so that the GO TO can be used only for exiting from a control environment. This has been done to enforce GO-TO-less programming.

The statements can be grouped into the following classes:

1. Descriptive statement .
2. Program structure statements.
3. Data movement and computational statements.
4. Control statements.
5. Input/output statements.

Descriptive Statement: The declare statement constitutes this class . A declare statement can come only in the beginning of a block. There are two types of Declare statements:

1. Declare Statement for structured variables: One statement of this type can describe one structure only. This restriction is imposed for clarity in reading and implementation ease.

```
DECLARE   1  JACK,
             2  AGE FIXED,
             2  WEIGHT FIXED,
             2  HEIGHT FLOAT,
             2  ADDRESS (1:10)  CHAR,
             2  MARRIED   BIT ;
```

Note that the structure consists of a major structure name and several elements. There are no minor structures.

2. Declare statement for non-structured variables: In this statement data types and storage classes may be attributed to a variable or an array. All the attributes of a simple variable or array must be given at one place only. This improves readability.

The precision of arithmetic variables is not to be specified because they are always given a standard precision by default. Absence of the EXTERNAL attribute implies the INTERNAL attribute. Elements having the same attributes may be factored together for convenience. The declaration of an array must include both the bounds.

```
e.g. 1.  DECLARE    A FIXED,  B FLOAT,  HOURS (10:40)
                    FIXED EXTERNAL, NAMES STATIC CHAR,
                    STATUS  BIT,  (ATR,  MATR(1:10))FIXED;
e.g.2.   DECLARE    A FIXED, B  FLOAT,  A STATIC;
```

This is invalid because all the attributes of A are not specified together at one place.

Scope rules: The scope rules for the declarations are the same as those of PL/I.

Attributes: The declare statement is used for describing structures and for assigning attributes to variables and structure elements. An attribute may assign a storage class or data type or range to a variable.

A check list of all the possible attributes which can be used in a declare statement is given below:

Storage class Attributes	:	BASED, STATIC
Data Type Attributes	:	FIXED, FLOAT, CHAR, BIT, POINTER
Range Attribute	:	EXTERNAL.

The default options in the three cases are AUTOMATIC, none and INTERNAL respectively.

Control Statements: The GO TO statement is for unconditional branching; the destination is specified by a label constant. The unconditional branching is restricted so that one may branch only to labelled END statement which terminates the block or group to which the GO TO statement is internal.

This is consistent with the GO-TO-less programming concept. The GO TO is provided only for the purpose of exiting from a block or a group.

Examples of usage of GO TO statement.

1.
```
           BEGIN;
           .
           .
           IF  A NE B  THEN  GO  TO XIT;
           .
           .
      XIT : END;
```
The branch to XIT will terminate the BEGIN block.

```
        DO ;
        DO    J  =  1   TO   10  ;
        .
        .
        GO  TO  TERMINATE;
        .
        .
        END;
TERMINATE : END ;
```

The iteration ends if the jump is made to TERMINATE.

3. For an example of termination of active procedure by the use of a GO TO statement refer to the termination of procedures in section 3.4.1.

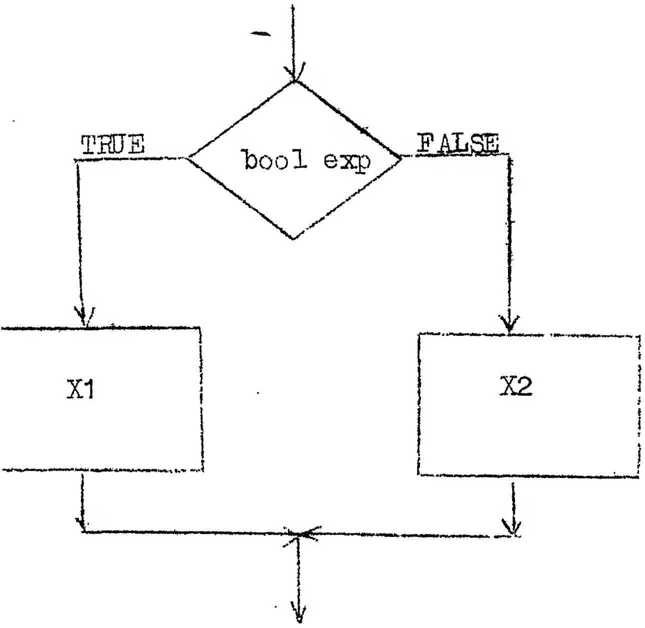
The IF statement : The IF statement is identical to the Dijkstra construct for selection from a pair, discussed in Chapter 2. They are reproduced below for convenience. The IF statement is one of the important constructs which is an aid towards a good program structure.

The IF statement construction is as follows:

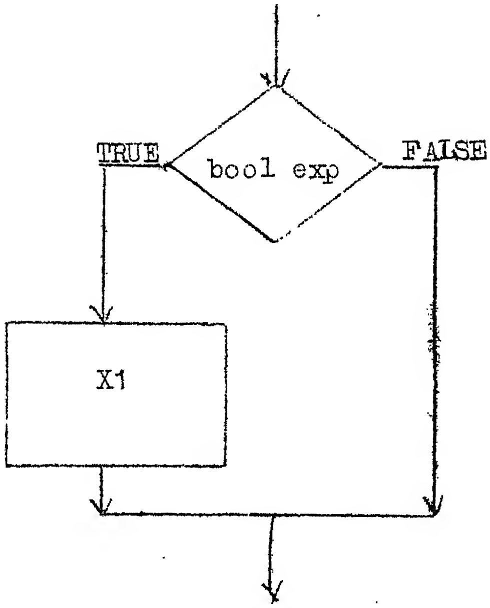
```
IF  <bool.expr.>  THEN  X1 ; ELSE  X2 ;
```

where <bool.expr.> is a condition i.e. boolean expression, <X1> and <X2> can be either a statement, group or begin block.

The ELSE clause is optional. The syntax rules for forming an IF statement and for nesting are the same as in PL/I. The meaning of the IF statement becomes clear from the diagrams given below.



'IF THEN ELSE' STATEMENT



'IF THEN' STATEMENT

<u>The DO statements</u>: The common uses of the DO statement are to specify that a group of statements is to be executed iteratively under count control or as long as a given condition holds true, or it may be used to delineate a group of statements for control purposes,. Consequently there are three DO statements for each of these three functions.

1. <u>Count controlled DO statement</u> : It has the following syntax.

DO <variable> = <initial value> TO
<final value> BY <step> ;

or DO <variable> = <initial value> To <final value> ;

The <initial value> , <final value> and the <step> are

restricted to either constants or variables.

The DO statement can be used in the same way as in PL/I. Note that we can do away with this type of DO statement and manage with a DO WHILE statement (described below); but since count controlled iterations are so common it has been provided as a convenience.

An example of use of a count controlled DO statement:

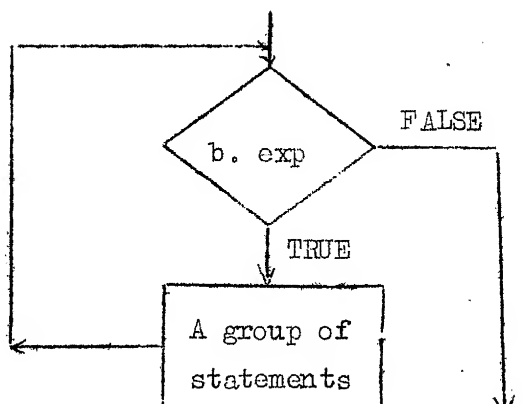
```
DO     J = 1  TO  10 ,.
:
END,.
```

The statements between DO and END will be executed 10 times-unless the loop is terminated prematurely by a GO TO statement .

2. <u>DO WHILE statement</u> : It has the following syntax and can be used in the same way as in PL/I.

DO WHILE <b. exp> ,. where, <b.exp> is a boolean expression.

The DO WHILE statement provides us with Dijkstra's pre-checking loop construct shown below.



'DO WHILE' STATEMENT

Example of a DO WHILE statement :

```
DO    WHILE    X  GT  10 ;
:
END
```

The statements between the DO and END will be executed as long as X is greater than 10.

3. Non-iterative DO statement : It is simply the statement 'DO, .'

This DO statement, in conjunction with a DO group is used to indicate the DO group is to be treated logically as a single statement. It has another very important function to perform in conjunction with the GO TO statement- that of providing means to terminate the execution of a count controlled DO group or a DO WHILE group without completing the iterations. This can be done by nesting the iterative DO group within a non-iterative DO group. When the iteration is to be terminated it can be done by an unconditional jump to the END statement which corresponds to the non-iterative DO group. See subsection on 'GO TO statement' for an example.

The DO statements can be used exactly as in PL/I; keeping in mind the restricted nature of their syntax in MINIPL. The rules for nesting of DO groups in PL/I are valid in MINIPL also.

The count-controlled and the DO WHILE statements of MINIPL cannot be combined into one statement as in PL/I. The combination is rarely used and therefore has not been provided.

I/O statements: Only stream oriented I/O has been provided for it's simplicity. Even in this class the data-directed I/O has not been provided because it's utility is questionable and it is so different from the other modes of I/O that the beginner will have difficulty to switch from the data-directed I/O.

To keep the I/O statements as simple as possible the FILE option has been excluded because the beginner will only be concerned with the standard input and output files. The repetitive option, too, has been excluded; and, since the repetitive option is excluded there is not much point in having array input and output. It does not reduce the I/O capabilities. The repetitive option can be done by the use of a DO loop.

The format has been simplified by allowing only integer constant for specifications of field widths and counts. The less often used LINE, and COLOUMN control format items have been eliminated. Only immediate formats are allowed -it aids in better readability than the remote format.

Although MINIPL has no character string data, the output statement can specify a character string constant in the output list. To avoid conceptual confusion, since character string data is not there in MINIPL, we shall call it message string instead of character string. The message string has been provided to serve the same purpose as the hollerith fields in Fortran formats. Although the same effect could have been achieved by

using the character data - only it would be very cumbersome. The message string is output in the same way as a character string is output in PL/I.

Some typical I/O statements are given below:

```
GET LIST  (X, A.B, C(B)) ,.
PUT PAGE LIST  ('OUTPUT  IS = ', B),.
PUT SKIP (3)  LIST (A*B + 6, 6.3 - 4*Y),.
GET EDIT (A,B,C)  (E(10,3),X(5),F(6), SKIP(3), A(1))
PUT PAGE EDIT ('PRODUCT IS ' = ' , A * B, 'SUM  IS = ', A + B)
                 (2 ( A(1),  X(2),  F(4))).
```

Data movement and computational statement :

Internal data movement involves the assignment of the value of an expression to a specified variable . The expression may be a constant or a variable or it maybe an expression which specifies that computation is to be made. The assignment statement is used for internal data movement as well as for specifying computation. The general form of the assignment statement is :-

Variable = an expression ,.

The expression is evaluated and assigned to the variable. The value of the expression will be converted to match the data type of the variable. The conversion rules are the same as those of PL/I.

3.4.6. Expressions :

PL/I has very generalized expressions. Mixed mode expressions

are allowed to the extent that a boolean variable may be multiplied by a decimal variable or a relational expression may be evaluated and added with another relational expression. This involves conversion of data types from one type to another according to fixed rules. While forming the expression the programmer has to keep track of the data type of the result after each application of an operator. This is very prone to errors and there is a possibility that the programmer may unintentionally mix the data types in an expression which will be difficult to debug. For the sake of clarity MINIPL disallows mixed expressions altogether; consequently there are three categories of expressions- namely Arithmetic, Relational and Boolean expression.

Arithmetic expressions are formed in the usual way by using the operators + - * / ** and the paranthesis. Variables and constants used as operands must be of decimal data type. Mixed arithmetic expressions involving floating point and fixed point data is not allowed. A simple variable or constant of decimal data type can also be considered to be an arithmetic expression. The value of an expression is of the same base as the operands.

Relational expression :- A relational expression may be formed by

(1) Using any relational operator on two arithmetic expressions.

e.g. A + B GT C/D is a relational expression.

(2) By using the operators '=' or 'NE' on character variables or constants.

e.g. Z NE 'C' is a relational expression

where Z is a character variable.

3. By using the operator '=' or 'NE' on Boolean and relational expressions. If a relational expression is used as an operand it must be paranthesised.

e.g. 1   A AND B (bool.exp) = C OR D (bool.exp.) is a relational expression.

e.g. 2.   A AND B (bool.exp) = (C LT D) (relational expression)

Paranthesisation is necessary to remove ambiguity.

e.g. A = B NEC C

(where A,B and C are boolean variables)

can be interpreted as (A = B) NE C

and also A = (B NE C)

<u>Boolean expression</u> : A boolean expression can be formed by using the boolean operators on boolean expressions and relational expressions. A boolean variable or constant is also considered to be a boolean expression. The order of evaluation is from left to right with 'NOT' having the highest precedence and 'OR' having the lowest precedence. Relational operators have lower priority than boolean operators. e.g. of boolean expressions:

1. A OR B
2. A OR NOT B
3. A OR B AND C
4. A AND X GT Y .

3.4.7. The Basic Elements of MINIPL

MINIPL uses the 48 character-set. This implies that many of the operators and delimiters will have to be represented by a combination of characters.

Eg. '>' is represented by 'GT'
'; ' is represented by ',.'

The representation of the semicolon causes some problems. Consider the statement PUT LIST (A,. 3);
Comma followed by the arithmetic constant .3 leads to the formation of a semicolon. The user has to be careful to leave a blank between the comma and the period in such situations.

Identifiers : The syntactic rules for creating an identifier are the same as in Fortran. The maximum length of an identifier, however, may be larger than six and depends on the implementation. Any two identifiers in a MINIPL program must be separated by at least one blank or some other delimiter.

Reserved Words : Many identifiers are reserved and may not be used for any purpose other than the one for which it is intended. A list of reserved words appears in Appendix A.3.

Use of Blanks : Blanks may appear anywhere in the MINIPL program as long as they are not contained in identifiers, constants (except character string constants) and composite operators.

Comments : Comments are permitted wherever blanks are allowed in a program. (except in character string constants). The general format of a comment is.

/* character-string */

In the last few pages a description of MINIPL has been given very briefly. Since it is not possible give a full description in such a small space there are bound to be some omissions. In case of doubt, the final arbitrator is the grammer given in Appendix A.1 for syntax, and the PL/I manual (19) for the corresponding semantics.

3.5. Conclusions:

In this chapter we have come up with the specifications of a language based on the language design principles expounded in Chapter 2. Not every one will agree with the decisions taken because of variations of opinion. Because of lack of any formal or quantitative methods of testing, only time and usage can show how good a language MINIPL is for the beginner.

CHAPTER 4

IMPLEMENTATION : DESIGN CONSIDERATIONS AND OVERVIEW

Starting from a statement of needs and through a discussion of language design considerations in the context of these needs, we came up with the specification of the language MINIPL towards the end of last chapter. Now onwards, we shall concentrate on what approach we have taken in designing certain major parts of the compiler for this language.

In the present chapter, after looking at the translation process in general in section 4.1, we discuss the choice of the technique for translation in section 4.2. Section 4.3 discribes the various languages involved in the translation process. In section 4.4 we discuss the choice of the number of passes and in section 4.5 various aspects of m/c. independence. Through these sections we indicate the major design decisions. In section 4.6 we discuss the implementation of these decisions in terms of present implementation. The last subsection 4.6 gives a description of the overall organization of the compiler and points to where the details of individual tasks and associated routines are to be found. We close with a discussion of how to incorporate the compiler into the operating system.

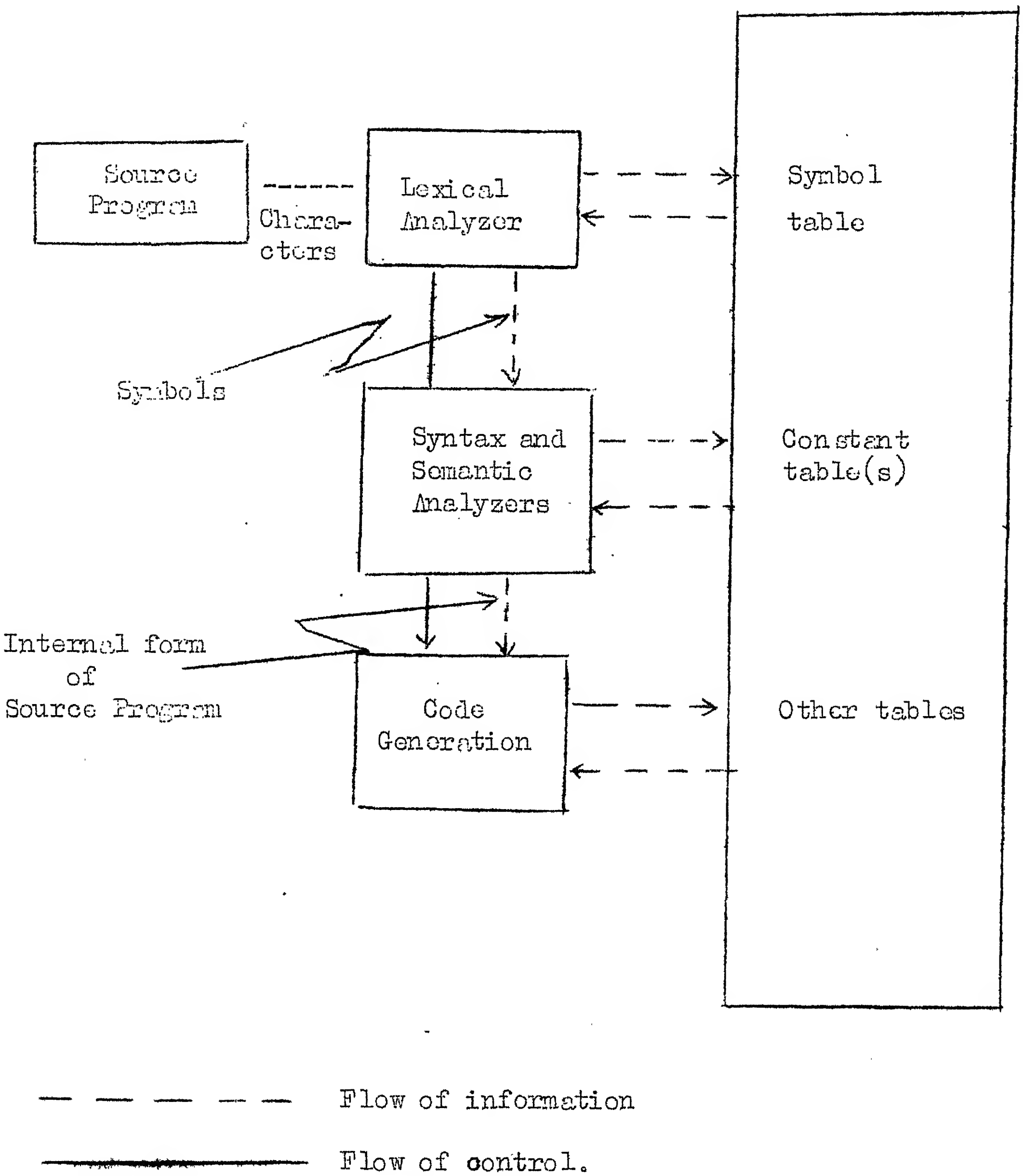

Source Program
Characters
Lexical Analyzer
Symbol table
Symbols
Syntax and Semantic Analyzers
Constant table(s)
Internal form of Source Program
Code Generation
Other tables
Flow of information
Flow of control.

Fig. 4.1

4.1. The Compilation Process :

A compiler basically performs an analysis of the source program and then a synthesis of the object program. To get a general idea of the process fig. 4.1 may be helpful. The figure is just for illustration to show the basic tasks involved; the distribution of work is quite often not so rigidly fixed as indicated by the boxes. The flow of information lines may also not be what they are indicated in the figure.

Primarily, the lexical analyser takes in the characters of the source program and builds the valid symbols of the program. It does blank squeezing and may do comment deletion. Many times it produces it's output in a fixed length internal code, making the task of the rest of the compiler simpler by letting it operate on fixed length code rather than the variable strings of characters.

Syntax and semantic analysers disassemble the source program into it's constituent parts, building the internal form of the program and putting information into the symbol table and other tables. A complete syntax and semantic check of the program is also performed.

The last phase corresponds to the actual translation of the internal source program into the machine language. Some times a code-generation preparation box is added before the last one. Certain tasks are specifically sealed in this box - namely run-time storage allocation to variables, and some optimization on internal form produce better code.

Having got the general idea, it must be noted that the above is a sketchy description of the logical connection of various tasks. The different tasks can be performed one after the other or in a perfectly parallel interlocked fashion. Between the two ends, other organizations also emerge. Similarly the tables and the information kept may very much depend upon the language and the objectives of implementation. Many times the details of information which it is required to maintain may be come clear through the implementation only.

We shall now discuss the choice of the basic strategy for our implementation especially in view of the requirement laid down at the end of chapter one.

4.2 Syntax Directed or Otherwise :

Various techniques have been used for translating programming languages. Among these are statement categorization, keyword recognition or template matching, syntax direction and table direction. Till early sixties,most of the compilers for high level languages were of the brute-force categorizing a new statement by a type; then, by a rather complicated process emitting code that represents the semantics or meaning of the source statements. These compilers sufferred from the lack of a central underlying concept and tended to be disharmonious collection of routines joined together in an ad hoc fashion. With the development in the theory of languages there emerged a formal notation of specifying

the rules of well formed sentence construction (syntax) and the methods that use this description to recognize and parse the sentences of the language. These methods of translation tend to be independent of the particular language to be implemented as they use a 'tabular' representation of 'syntax' or the rules of forming the sentences of the language. This separation of the description of structure and analysis algorithm is in contrast with ad hoc-techniques where the structure is in an indirect fashion incorporated in the parser itself. This makes the modification of the input language simple as not much change is required in the parsing algorithm proper. Also the algorithm provides a central underlying concept which coordinates the whole translation process in a natural way. Once the parse is through or while parsing, 'semantics' or the meaning of syntactical entities may be determined. This corresponds to the production of machine code corresponding to high level language constructs. This latter process,however, is mostly carried out by specific algorithms embodying most of the information about language semantics. Recently efforts have been made to describe semantics also in a formal machine-independent notation but few, if any, practical compiles based upon these have been implemented. Nevertheless, the central syntax directed technique does make the task of introduction of semantics easy and uniform.

The choice thus obviously is that of a syntax directed technique. A recent effort (18) attempted a compiler of PL/I on 7044 using the statement categorization process. Syntax analysis in the PL 7044 (18) is a decentralized process and is carried out by several statement decoding routines. This strategy, it is claimed, has the clear advantage of being fast when compared to a centralized syntactic analysing scheme. The advantage is said to accrue from the fact that each decoding routine not only checks the syntax but also assigns the semantic interpretation simultaneously, a possibility which is precluded in the case of centralized scheme. This conclusion, however, does not seem to be well founded. The reason of simultaneity seems to have nothing to do with a centralized technique. As we shall see, the semantic analysis can be (and is in the present implementation) done hand-in-hand with the recognition of individual syntactic entities and there certainly is no repetitive checking performed, as suggested. The ad hoc checks which are inevitable in the type of analysis in (18) not only do harm to the program clarity but also contribute to repetitive coding which is a disadvantage, as conceded by the implementors also (18). The statement categorization type of syntax check requires lot more documentation than what is needed with a central technique (hardly any, except for the syntax description and the basic algorithm). We must however confess that our choice was in a great measure facilitated by the presence of a constructor (23) to automatically

build tables directing the syntax analysis process.

4.3. Languages involved in the Process :

The source language-let us call it L_o-is of course MINIPL. The ultimate target language L_t into which this is to be translated is the m/c language of the computer on which MINIPL is to be implemented. The intermediate languages can be denoted as $L_1\ L_2\ \ldots\ L_{t-1}$. Let us call the three languages involved in implementing MINIPL , L_o' , L_i and L_t' to bring out their relationship better. The whole translator may be regarded as a transformation from L_o to L_t. We may write,

$$T(L_o) \rightarrow L_t \qquad (1)$$

The relationship between the various phases of translator is through the intermediate languages and tables of information they output and accept.

In particular the language L_o' will be very close to L_o (a coded representation in fact). The language L_i will be the main intermediate language chosen to be the Gries Quadruples (16), in the present case. We may have a further language L_t' as the assembly language of target computer. (though this is not considered an efficient solution).

Now the mapping T can be considered as a combination of four submappings, T_s, T_1, T_2 , T_3. The process may now be represented as a sequence of transformations :

$$T_s \ (L_o) \rightarrow L_o'$$

$$T_1 \ (L_o') \rightarrow L_i$$

$$T_2 \ (L_i) \rightarrow L_t'$$

and $$T_3 \ (L_t') \rightarrow L_t$$

T_s evidently is the lexical analyser, T_1 the syntax and semantic analyser and T_2 the code generator, T_3 is a conventional assembler, the need for which may or may not be there depending on the level of the intermediate language L_i.

4.4. Choice of the Number of Passes :

By the number of passes in a compiler, we mean phases which produce intermediate output in core or on output units. The successive phases are either overlaid on the previous ones and use the information in core, or are loaded as separate programs which read the input form the input devices. The choice of number of passes is dependent upon many factors, among them : The core size. How fast the compiler should be ? How fast should the object program be ? How much debugging facilities should the object program have ?

In general, if a lot of optimization is desired, the core size limitation will dictate that the number of passes be greater. On the other hand fewer the number of passes, the lesser the quantum of I/O activity, in general, and to that extent the compilation is faster. Sometimes the sheer size or the nature of source language

itself makes it imperative that more passes be employed.

In the present case it was decided to make the compiler essentially one pass, not counting the intermediate-language (L_i) - processor which will be discussed soon. The reasons to justify the choice were :

a. In the kind of environment envisaged (of beginning 'programming' students) fast compilation is more important than highly optimized code.

b. The present language MINIPL is a heavily reduced version of PL/I and as such the size problem is considerably less acute.

c. One reason that makesa one pass compilation impossible, i.e. use of variable before they are declared, does not hold in case of MINIPL because of the requirement of explicit declaration for all variables in the beginning of a block. The other problom (18) of not knowing anyting about the formal parameters of a procedure is alleviated by the following features of MINIPL. The declarations will immediately folow the procedure heading. Also labels or procedure names can not be the arguments of a procedure. That still leaves forward references of go-tos and calls (essential because of recursion) but these can be taken care of and the method is described when handling of labels is discussed in chapter 7 (Part I).

d. Larger number of passes imply a larger number of interfaces. In the present situation for a closely knit team of two, it seemed more advantageous to have a single pass rendering the control simpler.

4.5. Machine Independence :

By a machine independent implementation we could ideally mean a compiler that can be used on any machine without modification. Such a definition is indeed preposterous, especially for a program like a compiler whose task is to map the source language (L_o) onto the machine language (L_t) of the machine. However, it is obvious that most of the logic of the compiler including syntax analysis and building of syntax trees is essentially independent of the target language L_t. The reasons why it has not been possible, in many cases, to avoid the waste associated with repetitive coding of the aforementioned logic, are :

a. This logic is inextricably mixed with that of producing object code to make the reuse of coding nearly impossible. The advantage claimed by Gupta et. al.(18) of fastness accuring by assigning semantic interpretation (target language, L_t, oriented) is a hurdle in achieving machine independence.

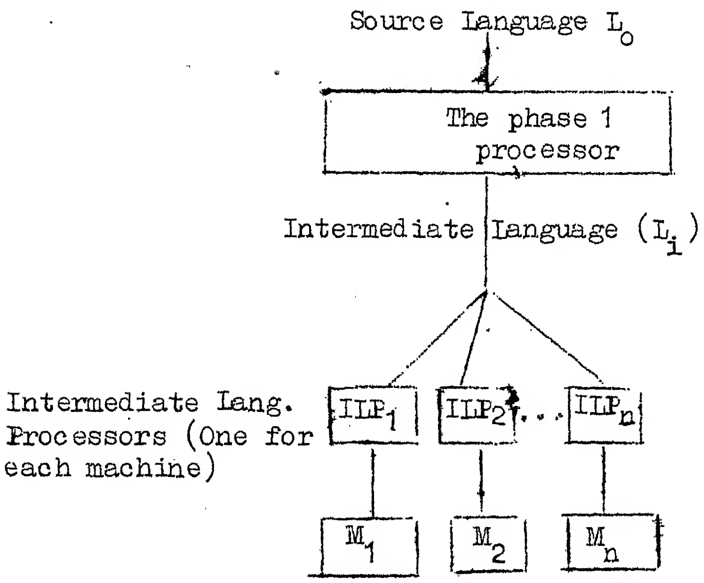
b. The choice of the medium of writing the compiler has been the assembly language of the computer on which implementation is being carried out. This feature, main reason for which

is the efficiency of execution time for compiler, makes the computer immobile.

We discuss next how to get around the above two obskcles to machine independence especially in the context of present implementation. The solutions also solve some other problems which will also be indicated.

4.5.1. The Language L_i :

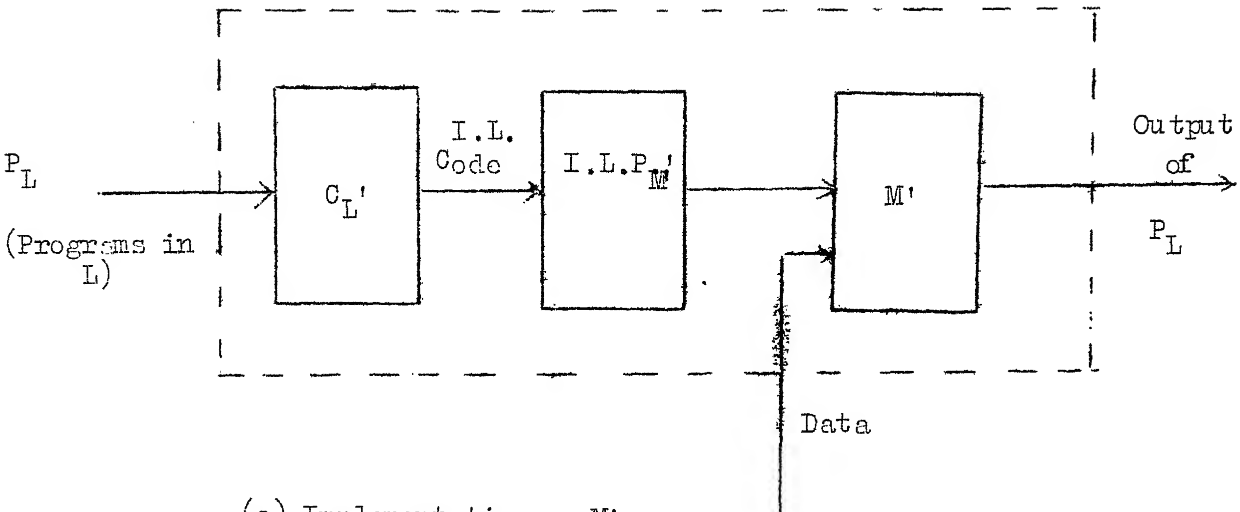
An obvious method to separate the machine independent portion from the machine dependent one, logicwise, is to separate the work in two phases. The first one produces it's output in some intermediate language (L_i) and the next phase processing this. Thus the compiler could be implemented, assuming that the language in which it is written can be executed on the compiler - how this done will be described in the next section, on different machines (M_1, M_2, ...) as shown in fig. 4.2.



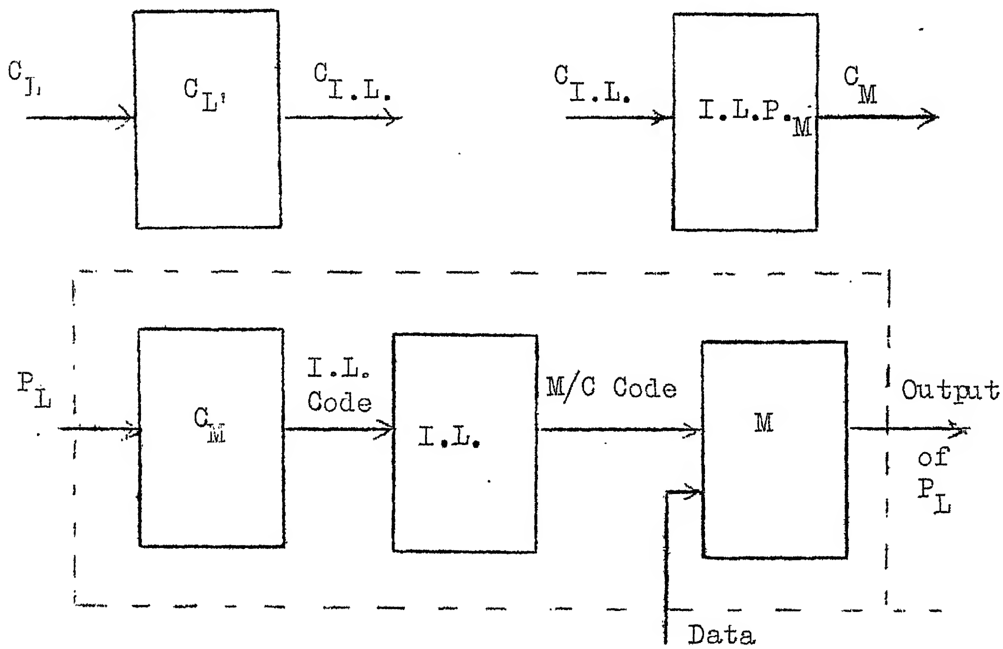
The main question to be answered is, what should be the level of detail in the language L_i. It could be quite high level incorporating the control constructs of the high level languages or it could contain simpler and more elementary control flow mechanisms. The type conversion operations could be incorporated here or this could be done later. In general, higher the level, probably more independent the phase I processor will be. However too high a level for the language L_i will just be transfering the problem to this level. In the present implementation it was decided to do most of the processing in phase-I processor, here on referred to as the 'compiler', so that the intermediate language processor is simple enough to be implemented with a minimal effort. It is this decision that justifies calling the present compiler one pass. The second pass is - as will be seen shortly - equivalent to the conventional assembly where, the output of the compiler which is in the assembly language, is assembled.

4.5.2. Making the Compiler Available on Different Machines : Portability

Even when the machine independent logic is separate from the machine dependent one, to make the compiler usable on different machines it is essential that the language in which we have written the body of two compiler is available or can be easily made available on a given computer. The approach taken in the present implementation is (fig. 4.3) :



(a) Implementation on M'



(b) Implementing L on M.

<u>Notation</u>: C_s : Compiler for L written in the language s.

$I.L._s$: Processor for intermediate language written in the language s.

P_s : Arbitrary programs written in the language s.

M & M' : Machine language interpreter for machine language M & M'. (These could be the machine themselves).

Fig. 4.3

Take a language L' which is available on machine M'. Write the compiler for L in L'. This compiler $C_{L'}$ generates the intermediate language code. As soon as we have written the intermediate language processor, we have the language L implemented on M' (fig. 4.3a). The main aim however is to make available L on a machine M which may not have the language L' available on it. We proceed as follows(fig. 4.3b) :

Code the compiler for L in L itself. Now using the compiler $C_{L'}$ (written in L') translate the new compiler C_L. We have a compiler $C_{I.L.}$ written in the intermediate language. This compiler can be implemented on any machine as we assumed that I.L. is implementable on any machine without difficulty. Thus we have the compiler for L written in the machine language of machine M.

The version of compiler written in L can be effectively used for maintainance, that is improvement and additions. The new compiler in intermediate language can now be obtained by using the old running version of compiler to translate C_L.

4.6. Present Implementation :

Having talked of the two major requirements and their influence we shall now give the important design decisions while describing the present implementation. Pointers to where the details are to be found in next chapters will also be given in this section. First we describe the intermediate language chosen, then consider the language(s) for the coding of compiler and their

limitations with respect to machine independence. This will be followed by a brief description of the overall control flow in the translator. In this last subsection we shall also talk about how the compiler could fit into the operating system.

4.6.1. Quadruples :

The intermediate output from the compiler in the present implementation consists of Gries Quadruples (16). Primarily these have the form :

operator, operand 1, operand 2, operand 3

where, 'operator' is the binary operator operating upon operand 1 and 2 and storing the result in operand 3. This form fits in naturally in most arithmatic and similar operations where the operands are locations. For other operations ~~,~~ these could be labels or procedure names etc. Many Quadruples may have fewer operands than the number shown. A list of Quadruples can be found in Appendix G . This is by no means complete as new quadruples may be specified to take care of a specific situation as semantic interpretation of more and more syntactical entities is incorporated.

An important thing that should be pointed out is that the level of our intermediate language is quite comparable to an assembly language and thus their processing should not be very much more difficult. In fact, in line with the philosophy of doing the maximum amount of work in the compiler phase (section 4.5.1) making implementation of Quadruple Processor simple, it has been decided to do

the storage allocation in the compiler itself. Also most of the table maintainance is already done to minimize repetition in Quadruple Processor. Storage allocation and table management are discussed in chapter 7 and how the Quadruple Processor can be made simple using these features is also discussed there.

<u>Some Advantages</u> :

The operands in general are locations and thus action of an operand is to produce a result in a location. The interaction in Quadruples is strictly through locations. This is in contrast to triples (op, operand 1, operand 2) (16) where the result is assumed to be associated with the triple itself. This makes the interdependence between triples strong and their movement to facilitate code optimization more difficult. The detailed discussion may be found in chapter 11 (16). Another advantage of quadruples is that all operands being locations, no reference is made to any registers, accumulators etc.,features which are dependent upon the architecture of the individual machine. This makes the generation of quadruples easier, though a lot of temporaries get allocated in the process. The operands depend very much on the addressing scheme and for the block structured languages they may be collections of subfields indicating block count, offset, type, indirect addressing etc. The operands may be labels too. Exact details of these are given in chapter 7 while discussing the addressing scheme. The quadruple format may be varied to suit different possible implementations

of Quadruple Processor. To this end, a separate Quadruple generating routine is written which may be modifed to produce different formats. Details of this are given in Chapter 7(Part II).

4.6.2. The Coding Language :

Main body of the compiler has been written in FORTRAN (corresponding to the language L' of section 4.5.2). From the criteria of readability as well as the ease of coding this was more suited than MAP, the assembly language of the machine M' (IBM 7044). Some of the routines requiring packing, unpacking and shifting etc., have been coded in MAP. The interface is well defined between MAP and FORTRAN and hence does not pose much problem. Also it is possible that the Quadruple processor may require information in the reverse order from the one in which it was generated by the compiler. This may include header information available only at the end of compilation of an external procedure. An easy access to such information is possible if a sufficiently large number of secondary storage files are available. FORTRAN meets this need to a certain extent.

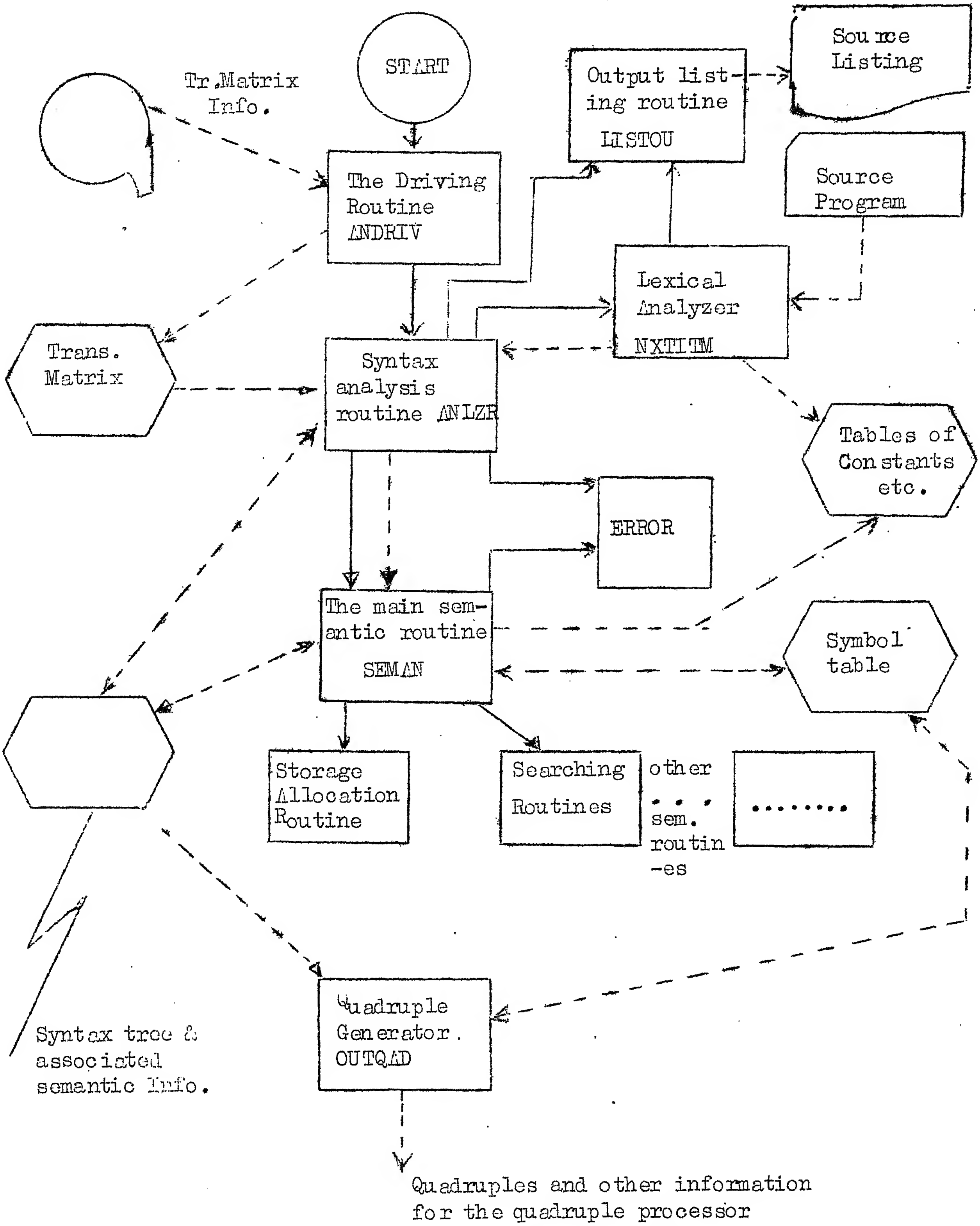
The language, L', (Fortran here) is however not of much significance as far as machine independence is concerned, since after the initial implementation on M' it is discarded in favour of MINIPL. Thus the question becomes whether the above mentioned tasks can be performed in it. The problem of packing and unpacking can not be tackled in the present MINIPL. If variable ranges for integer variables are added it might become possible. Another thing

that MINIPL lacks is file handling. We have in MINIPL, only one input file and one output file. At the minimum we need an output file for source listing and one for quadruples. Thus MINIPL's ability to control secondary storage has to come from the machine language routines, written for each new machine M. This is what makes the MINIPL compiler less than portable. However, identifying the machine dependent routines and writing them as external procedures, will alleviate the problem to some extent.

Although PL/I does have a full complement of file handling capabilities etc., these were not included as this would have made the implementation complex and the compiler impossibly large. Moreover, file handling did not come within the perview of MINIPL whose main intent is to serve as a medium-of-instruction language and not necessarily as a systems programming language.

4.6.3. Organization of the Compiler :

Fig. 4.4 gives the flow of control and information in the MINIPL compiler at a very-very macro level. The term 'compiler' as pointed out earlier will be used to refer to the phase I of the implementation which produces as it's output, the intermediate language code(Quadruples). Phase II which is to produce the final machine code from Quadruples is to be referred to as Quadruple Processor. The desirable features of it to some extent will be outlined in Chapter 7. In this section after looking at the 'compiler' we shall also talk about processing of quadruples on IBM 7044 while describing the possible O.S. interface.



——— —— Indicates the direction of control transfer (procedure invocation)

- - - - - - - Indicates the direction of information flow.

(THE OVERALL ORGANIZATION OF COMPILER)

Fig. 44

The Flow of Control :

Fig. 4.4 shows that control is first passed to the main driving routine ANDRIV. This is the routine which is to interface with the O.S. For the present let us assume that only one job (or a series of job's, delimited by a $) is there on the input and quadruple modules for individual external procedures are to be outputted on the secondary storage. ANDRIV recognizes the main procedure heading and passes control to syntax analyzer. Syntax analyzer routine ANLZR now retains control until the end of the job is signalled by a $ when it returns control to ANDRIV which after completing the end-of-job formalities, if any, looks for the main procedure of the next job. ANLZR obtains lexical units from the lexical routine (NXTITM). performs the task of searching reserved word tables, building constant tables etc., in addition to forming of other atoms of MINIPL. ANLZR calls the main semantic routine SEMAN, where a checking of program structure is made and semantics corresponding to different syntactical entities carried out. Symbol table maintainance and storage allocation are also done here. Semantic information is associated with the syntax tree partly by the ANLZR (this is the information supplied by the Lexical, e.g. types, pointers to constant tables etc.) and partly by SEMAN. The quadruple generating routine OUTQAD is called from various places in SEMAN to generate quadruples corresponding to syntactic entities. ERROR routine is a centralized error messaging routine called with appropriate error code to print a specific message and take corrective action, if any.

Details of syntax analyser and program structure checking are given in chapter 6 (Part I), and of the lexical analyser and associated routines in chapter 6 (Part II). Other semantic routines are described in chapters 7 of the two theses (Part I and II). Storage allocation addressing and symbol tables are described in chapter 7 (Part I) and iutput output, and quadruplc generation in Chaptor 7 (Part II).

Operating System Interface :

The compiler as implemented and discussed above is only one link, although the most voluminous and complex, in the process of getting a MINIPL job run. To complete the process following things are required. The header information (for external linkages) and the quadruples are to be collected from the two secondary storage units and the information put on a third file, for every external procedure. The routine to do this must be called by the main driving routine. The output may be collected and then fed as data to quadruple processor which puts out the binary object program on another file. Quadruple processor returns control to the system which loads the object program in core and transfers control to it. A simple method to utilize the present operating system processor facilities on IBM 7044 is to delegate most of the quadruple processing work to the MAP assembler. The scheme envisages considering quadruples as MAP MACROS and writing macro definitions to generate code. Control can be passed to the MAP assembler by switching the

system input file to the unit on which quadruples were generated. One big disadvantage is that all the macrodefinitions (for all quadruple-types) have to be included in every MAP routine corresponding to an external procedure since there is no provision for defining system macros. Secondly MAP assembler is awkward for the present task as it can not make use of the storage allocation in MINIPL compiler phase. But the most important thing against this mode of implementation is it's slowness. A routine containing the three run time storage allocation macros, one macro for fixed point addition and five or six uses of the latter took almost a minute to assemble (Appendix E).

CHAPTER 5

TRANSITION MATRIX TECHNIQUE

The methods of parsing based upon syntax, come under the group of syntax directed techniques. Floyd (13) distinguishes between what he calls the 'syntax directed analysis' and 'syntax controlled analyses'. By the former he means use of analysers working on some direct representation of the syntax, while in the latter class the analyser is driven by some tables which are derived from the syntax. The general top-down analyser described by Floyd (13) comes in the former category. This and similar algorithms involve back-up and while they accept a wider class of languages (Floyd's algorithm is for an arbitrary CFL), they tend to be inefficient. Recent modifications have improved this drawback to a certain extent (9), but so far most compilers have used algorithms that parse more restricted (however, general enough to include most practical programming languages), class of languages, more efficiently. Most of these techniques are bottom up techniques, and need no back-up. The paper on translator-writing-systems by Feldman and Gries (12) contains a comparative study of several such techniques with respect to the following criteria :

(a) How much space does the recognizer use ?

(b) How fast is the recognizer ?

Certain variations were made in the constructor output in the interests of storage economy while maintaining the speed. This and some other variations are described in the present chapter. The main objective of the present chapter is to tell about the experience with the transition matrix technique. But before we can talk about these details, a description of the technique and the recognition process in short is essential. Greater details and formal proofs can be seen in (15).

5.1 <u>The input grammer</u> :

The input grammer is restricted to be an operator grammer, i.e., in no production should two non-terminals occur in adjacent positions on the R.H.S. This by itself does not impose any serious limitation but some breaking up and alteration is required, an idea of which may be gathered by looking at the operator grammer for MINIPL given in Appendix A.2.

The following is an example to illustrate the situation :

<decl non strc> → <decl non strc> <type>

<type> → FIXED / FLOAT

This is not permissible because of the O.G. restriction. The problem cam be tackled by making 'type' a lexical output unit TYPE. However, if previously <type> had included the case,

<type> → <initial list> .

<initial list> → INITIAL (...

⋮

rules in which <initial list> figures

⋮

then, INITIAL will have to be taken as a TYPE and <decl non strc> would percolate into all the statements where <initial list> has been used. The problem in the present case was solved by putting the condition that initialization can only come after all the attributes of the variables are specified. Thus allowing :

<decl non strc> → <decl non strc> TYPE <initial list>

The operator grammer is coded into numbers according to the scheme given in Appendix A.2 . The constructor reduces the number of symbols in the right hand side of a production to three. This simplifies the recognizer and allows a direct correspondance to be set up between states of transition matrix and certain non-terminals of grammer. The change does not essentially alter the structure of the parent O.G. but consists of inserting intermediate productions . For example,

<a expr> → <a expr> + <term>

will transform into,

<a expr> → <a expr - + *> <term>

<a expr - + *> → <a expr> +

and then into,

<a expr> → <a expr - + - term *>

<a expr - + - term *> → <a expr - + *> <term>

<a expr - + *> → <a expr> +

The newly introduced symbols are called Starred Non Terminals (SNTs) and are distinguished from the original Unstarred Non Terminals (UNT's)

by an asterisk. The new grammer is called the Augumented Operator Grammer (AOG).

By applying a sequence of rules (15) on the OG - the AOG is constructed systematically. The AOG has only the productions of the form,

$$U_1 \rightarrow U_2 , \quad U_1 \rightarrow U^* , \quad U_1 \rightarrow U^* U_2$$
$$U^* \rightarrow T , \quad U^* \rightarrow UT, \quad U^* \rightarrow U_2^* \quad U_1^*T, \; U^* \rightarrow U_1^* UT$$

5.2 Unique Parsing of an AOG :

Gries (15) has proved that if an AOG is unambiguous, so is the parent OG from which it was derived. The sufficiency conditions for an AOG to be unambiguous are listed below:

Condition 1.

For each pair U_i^*, T_j where U_i^* is a SNT and T_j a terminal, at most one of the following is true,

$$\exists \text{ a production } U \rightarrow U_i^* \text{ such that } U \in \mathcal{L}(T_j) \qquad (1.1)$$
$$\exists \text{ a production } U^* \rightarrow U_i^* \, T_j \qquad (1.2)$$
$$\exists \text{ a production } U^* \rightarrow T_j \text{ such that } U_i^* \in \mathcal{L}(U^*) \qquad (1.3)$$

where , $\mathcal{L}(X)$ is the left set of the symbol X, i.e., the set of symbols which occur, in some sentential form, adjacent and to the left of X . Furthermore, if (1.1) holds, there is only one such UNT U(SNTs U* in (1.2) and (1.3) are unique by construction).

Condition 2: For each triple U_i^* , U_1, T_j where U_i^* is an SNTs, U_1 a UNT, and T_j a terminal, at most one of the following is true:

$\exists$ a production $U \rightarrow U_i * U_k$, where, $U \in \mathcal{L}(T_j)$ and either $U_k = U_1$ or $U_k \overset{*}{\rightarrow} U_1$; (2.1)

$\exists$ a production $U^* \rightarrow U_i{}^* U_k T_j$, where, $U_k = U_1$ or $U_k \overset{*}{\rightarrow} U_1$ (2.2)

$\exists$ a production $U^* \rightarrow U_k T_j$, where, $U_i{}^* \in \mathcal{L}(U^*)$ and either $U_k = U_1$ or $U_k \overset{*}{\rightarrow} U_1$ (2.3)

Furthermore, if (2.1) holds, both U and U_k are unique while if (2.2) or (2.3) holds , U_k is unique.

If conditions 1 and 2 hold, a unique parse of the AOG is possible.

<u>The Constructor Output and the Parsing Process</u> :

The constructor checks for all pairs which of the three (1.1), (1.2) and (1.3) holds, and records the L.H.S. of the reduction and the subroutine S_{U*T} which reduces the prime phrase . Similarly, for triples, conditions (2.1), (2.2) and (2.3) are checked. If more than one condition holds, or the derivations from U's are non-unique- corresponding messages are printed.

The original Gries (15) scheme records the output in the form of a two dimensional transition matrix, with rows corresponding to SNTs and columns to terminals. The scheme makes use of a push down stack to hold the SNTs and a single location NL to hold the non-terminal U_1 (fig. 5.1).

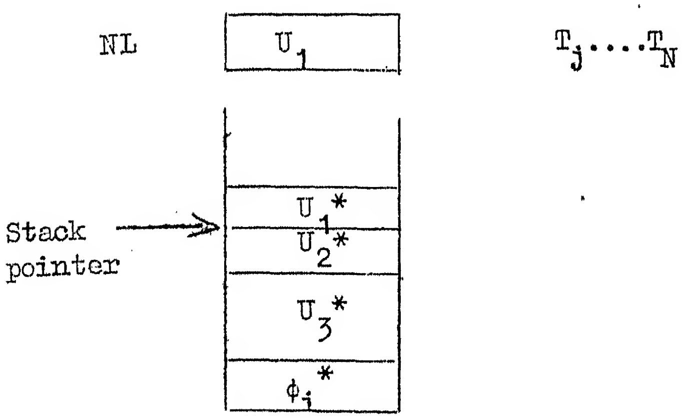


Fig. 5.1

At any stage the incoming terminal T_j and U_1*(top of stack) are used to find a subroutine S_{U*T} from the transition matrix. This subroutine, possibly through a series of if statements, checks if the U_1 is appropriate. If the process fails either in finding a subroutine in transitional matrix or while examining U_1, the incoming symbol T_j is in error and the statement is rejected. The process is started by putting is stack ϕ* corresponding to the terminal ϕ which is assumed to be appended to the left of all sentences.

5.3 The Three Dimensional Output Matrix :

The entries in the Gries matrix are sparse (around 800 in a matrix of approximately 65 x 50 , for the present implementation). Patil (23) devised a packing scheme in which each entry is in the following format (each entry is a 7044 (IBM) word) :

Bits	3	6	6	6	15
	///	NE	I	J	NSB

3	6	6	6	15
/////	0	0	J	NSB

NE : No. of entries in row I ; I : Row no. (code for some U*)

J : Column no. (code for a terminal); NSB : Number of the group of statements (corresp. to Gries routine S_{U*T})

Fig. 5.2

Searching consists of looking for positive NE's, checking I's and jumping accross the NE number of entries in case a failure occurs. The scheme was modified to a catalogue type (fig. 5.3) in the present implementation.

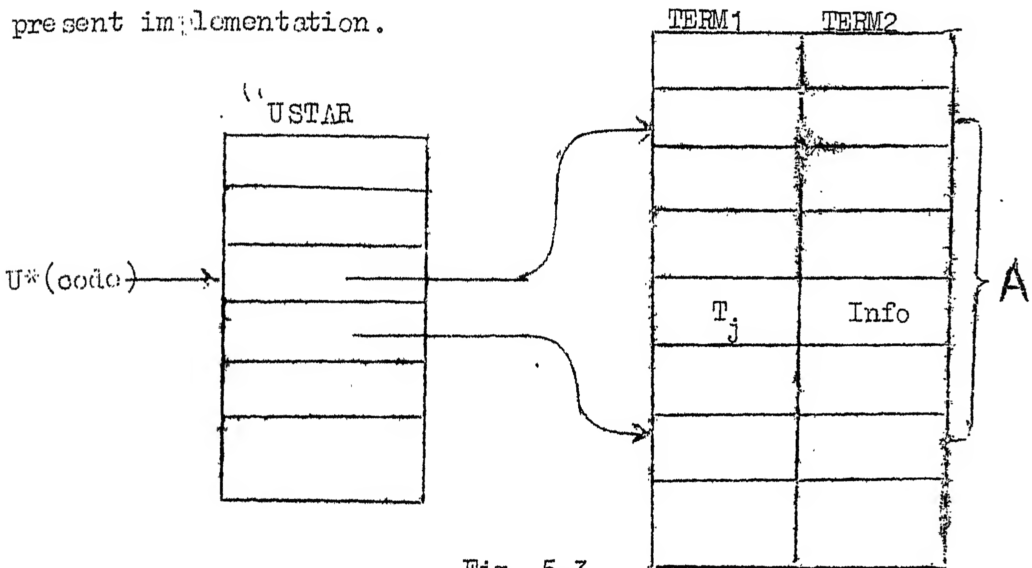


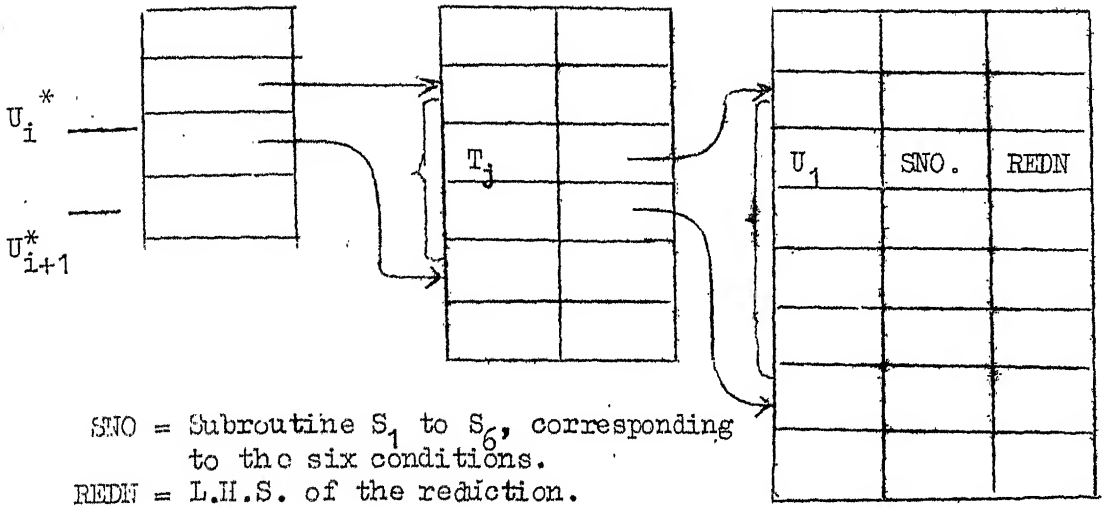
Fig. 5.3

Now the need for searching for U*'s is completely eliminated as the code for the U* can be used as the index to array USTAR. This entry gives the starting point in TERM1 where the terminals for the U* are stored. Thus A represents the range in which the terminal T_j is to be looked for. In case of a match the TERM2 entry will give the information about the reduction routine S_{U*T}.

Patil (23) has implemented Gries (15) groups of <u>if</u> statements by a sequence of 'GO TO's and checks, which examine U_1 and also call one of the six reduction routines to make appropriate adjustments in stacks and to build the semantic tree (to be discussed later). A typical group is :

```
NSB     IF (NL. NE. 0)    GO   TO N.1
        CALL  S1(U)
        GO TO 9999
N.1     IF (NL. NE. Ua)  GO  TO  N.2
        CALL S2(U)
        GO  TO  9999
N.2     IF (NL. NE. Ub)   GO   TO  N.3
        CALL  S5(U)
        GO  TO  9999
           :
N...    CALL ERRR (NSB)
        GO  TO  9999
```

The logic behind the generation of GO TO 's is not very clear. A unified scheme where U's form the third dimension is conceptually clearer. However, as the matrix size might have become very large, writing GO TO's for only the pertinent U's is a way out. However, if the catalogue scheme is extended to include U's (fig. 5.4), an economy in storage is possible. As for the speed, it remains essentially the same. Probably some marginal improvement may be there if binary search is employed which now becomes possible in the modified scheme. As for the storage each group now requires 3 words (3 bytes if byte arrays are used, as all the information is less than 255 (8 bits) when considered as a number) as against minimum 6 instructions (for IBM 7044) even if CALL is replaced by the setting of a variable I to subroutine number and using a 'computed GO TO' at 9999.



SNO = Subroutine S_1 to S_6, corresponding to the six conditions.
REDN = L.H.S. of the rediction.

Fig. 5.4

S.No. = Subroutine S_1 to S_6 corresponding to the six conditions.

REDN = L.H.S. of the reduction.

Fig. 5.4

5.4. The Syntax - Semantics Muddle :

A problem that occurs with the Algol 60 type of definition which we have adopted for our definition of different type of primarys(boolean, arithmetic, character etc.) is that all of them are ultimately defined as variables. This constitutes a violation of the rule that each derivation (in this case from <expr> to <variable>) be unique. A similar problem was faced by Wirth (35). The way to tackle this is that at the time a reduction is to be made a table search is done to see the type of the variable and the appropriate reduction made. This decision of applicability of a syntactic rule on grounds of essentially semantic information reflects the arguments that languages of the "Algol type" are not, strictly speaking, context free.

Another attempt to get around this difficulty was made by making different types of identifiers as terminal units and hand-over the entire type checking etc. to the lexical analyzer. This however, not only made necessary lot of flags etc. in the lexical analyser, rendering it clumsy, it also inflated the size of the grammer like anything. It was , therefore, decided to retain the inherent ambiguity and resolve it using semantic information about type.

5.5 Problem of Providing Enough Context :

Many times the grammer is not really ambiguous; still the recognizer fails because not enough context is used. For example, for a U*, T_j reduction no attention is paid to what is lying in the stack. The problem can be taken care of sometimes by introducing additional non-terminals (UNTs) and a few extra productions. The various situations are best described by a few examples of situations that arose during the course of the project. Sometimes it may be impossible to accomodate new additions to grammer, especially if the grammer that exists is to be retained without drastic modification. An example of how this problem came when it was decided to add initialization facility in the declarations, is given towards the end.

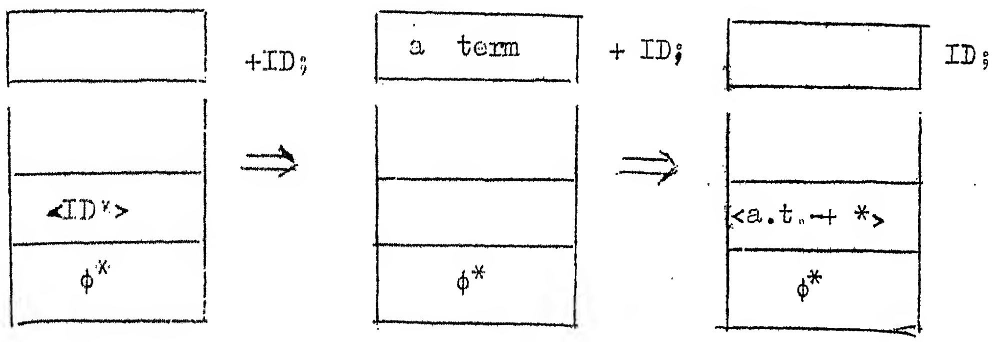
Consider the following productions:

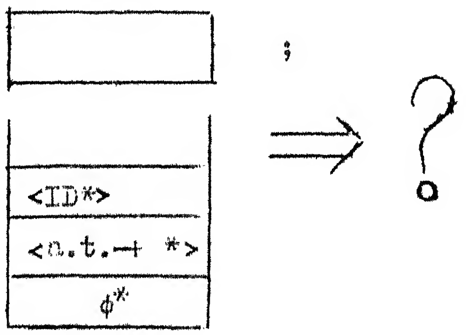
<a term> → <a term> + ID / ID (1) <goal> → <a term> ;

<b term> → <b term> Θ ID / ID (2) <goal> → <b term> ;

where Θ is some operator other than + .

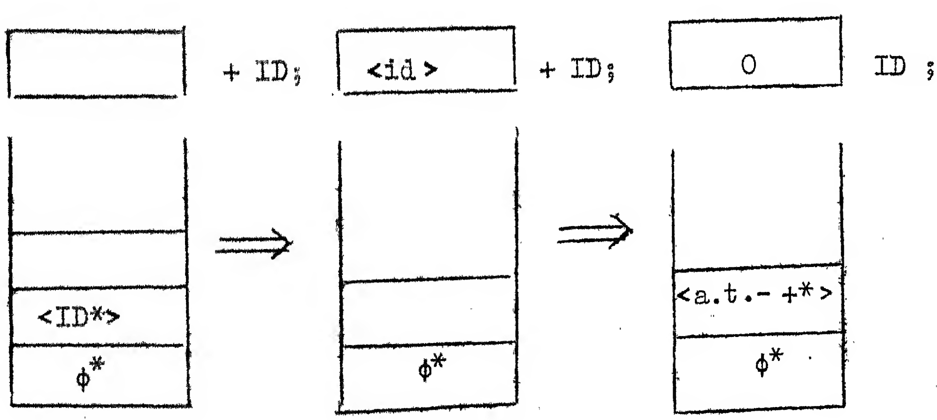
Now for the sentence ID + ID ; the reduction process will go as follows:

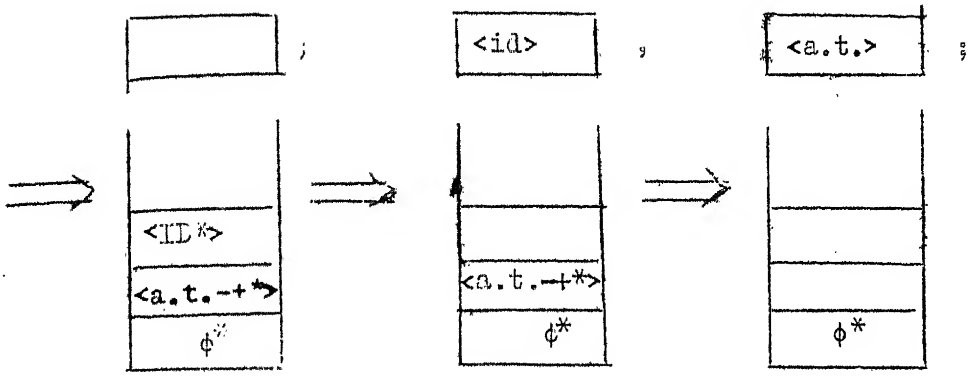




(ID* can go both into <a term> as well as <b term> which are both in the $\mathcal{L}$(S) of ;)

if we introduce <id> → ID ; and introduce <id> for every occurance of ID in (1) and (2) above, the problem is solved. The recognition process now is :





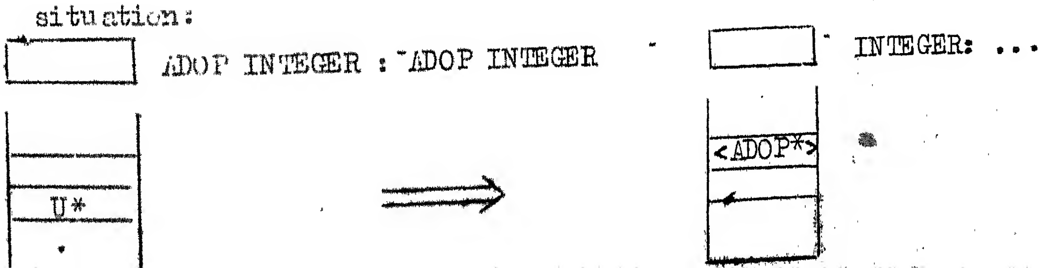
Here the formation of a U* U type of a prime phrase allowed taking advantage of the left context contained in U*.

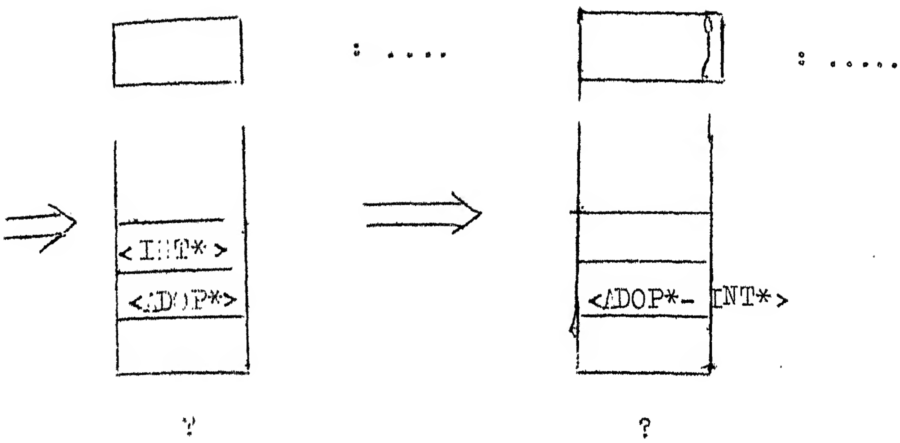
The above example is a simplification of the situation that occurred at many places. The details of actual production would have been unnecessarily cumbersome for illustration.

<u>Example</u> 2 :

<bound list>	→	ADOP INTEGER : ADOP INTEGER	(1)
<bound list>	→	<bound list> , ADOP INTEGER : ADOP INTEGER	(2)
<a expr>	→	<a expr> ADOP <a term>	(3)
	→	ADOP <a term>	(4)
	→	<a term>	(5)
<a term>	→	<a prim>	(6)
<a prim>	→	INTEGER	(7)

at a certain stage in the recognition process say we have a situation:





Both the reductions marked with '?' are possible because INT* is formed from (7) and it is indeed in the right set of <ADOP>* from (4). Also <ADOP* - INT>* is possible from (1). The problem arises because the context that can be made use of, is to the right. The previous type of solution of putting a common non-terminal does the trick here too. We can write ,

<bound list > → ADOP <integer> : ADOP <integer >

However the grammer given in appendix gives bound list element ::=<expr>:<expr> This not only makes unnecessary different sets of productions for<integer>.. <integer> , <integer >.. ADOP <integer> , ADOP <integer> .. <integer> , and ADOP <integer> .. ADOP <integer> .

During the semantics it can be ensured that<expr> is indeed of the variety desired. Such a solution is in any case unavoidable if we see the PL/I call by value. The simple variable surrounded by brackets is an <expr> too and thus an element of the call list is best taken as an <expr> - and the call by value assumed when <expr >

is of the type (<variable>). This point is of course not relevant to the problem of providing context but was mentioned in the passing, for the sake of completeness.

The solution previously described can not be uniformly applied. In Appendix A.2 it will be found that at some places <integer> is used and at others INTEGER, the terminal supplied by the lexical analyzer. The reason is obvious from the example of the rule:

<format spec1> → INTEGER <format spec2> (1)

Now the n.t. <integer> can not be used because of operator grammer restrictions. The only way to use it will be to put the three or four productions defining <format spec2> in place of rule (1), the solution being iterated if one of the rules has a non-terminal in the beginning on the R.H.S.

Another awkwardness of the solution illustrated by the example of <bound list> definition, where <expr> was used to define the bounds, is the untoward expansion of the transition matrix caused by the modification. In the previous case the change of the bound pair definition from INTEGER .. INTEGER to <expr> .. <expr> caused an increase in size of transition matrix by more than 25%. The increase is fairly irregular and could become critically large making packing of table entries essential even at the expense of speed. The problem could be more effectively tackled if the non-terminals introduced were at the lowest level,

so that no undesired symbols, which will have to be later weeded out in semantics, will be permitted by the grammer. That this is not always possible is shown from the next example.

An element of the initialisation list can be a signed or unsigned number. Character constant or boolean constant, if a definition based on the line of example of bound list is attempted, we have :

<initial list head> → INITIAL (<ini element> (1)

<initial list head> → <initial list head> , <ini-element> (2)

<initial list> → <initial list head>) (3)

<ini element> → ADOP <integer> | <integer> (4a)

ADOP <fltpt> | <fltpt> (4b)

At a certain stage in the reduction process we shall have,

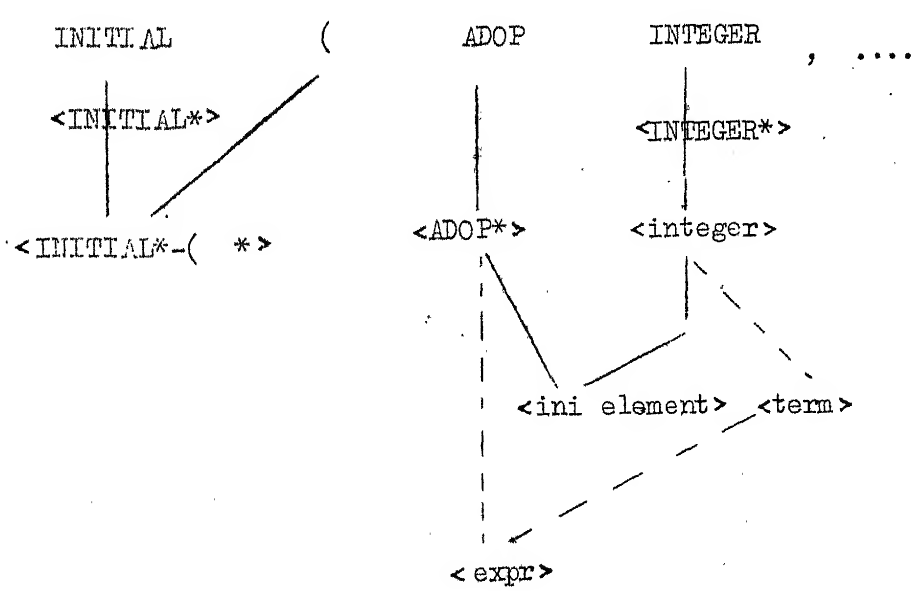


Fig. 5.6

Both the reductions (fig 5.6) indicated by the dotted as well as solid lines are possible, as U → U* U required U to be in left set of terminal ',' in this case, not bothering about the left context in<INITIAL* - (>*.

The problem can be overcome by defining the initial element as <expr> . If however a non-terminal at a lower level is desired, we may write:

<expr> → <unary-term>

<unary- term> → ADOP <term>

and use the new non-terminal <unary term> in the definition of <initial element>. Notice, however, that a non-terminal of the type <signed integer> → ADOP <integer>, is not possible as it can not be common non-terminal for <initial element> and <expr>.

Thus using <U-term> will necessarily allow lot of undesirable symbols in definition of initialization. In fact using <expr> will be better as it would not allow significantly larger number of symbol combinations while taking care of all the sub rules of rule (4).

5.6 Semantic Stacks :

In Patil's (23) scheme the recognizer has a secondary stack associated with the push down stack Gries' (15) algorithm uses in the recognition process. Another stack to hold the U's and T's forming the syntax tree was also incorporated.

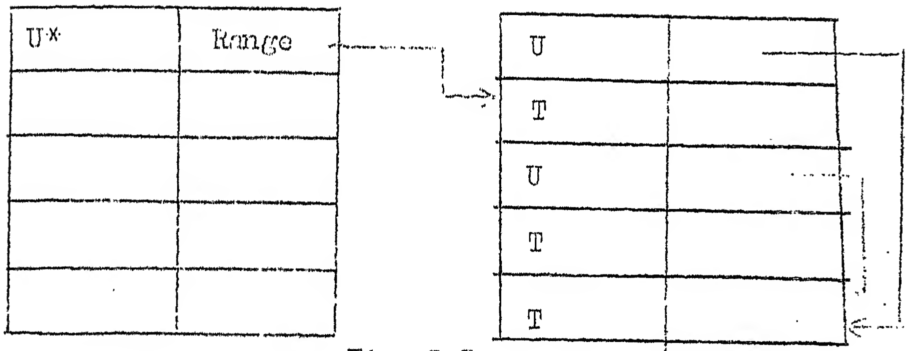


Fig. 5.7

As recognition proceeds U's and T's are pushed onto the stack and whenever a production with L.H.S. as a U is used, semantic routines are called with argument U. For each U pushed onto stack the range of the U is given opposite it.

However, as the semantics are associated with productions and not the non-terminals - a non-terminal may be on left of many productions - it seems more sensible to put the production numbers in the transition matrix. This is what has been done in modifying the constructor for use in the present work.

An array of left parts of productions is also generated which helps get the U to be used in the further reduction process. Though, to build a tree the ranges may be helpful, the information becomes redundant if we put the production number itself on the stack. As simultaneous processing by semantic routines will be assumed this has not been necessary. The range is not maintained but can be incorporated very simply.

In addition to the six groups of statements three more have been added. The detailed description can be found in

The first, i.e. group 7 , is to take care of the formation of the final goal non-terminals (though there may be only one distinguished symbol according to the definition of phrase structure grammer (15) , this can be achieved trivially by adding a few more productions in the grammer, deriving the present goal non-terminals from the distinguished symbol). The final statement (Appendix A.2) is of the form :

<statement> → <specific statement> ,.

Just before the goal non-terminal is to be formed, we have , < ϕ* - specific statement -; >* in the stack. A new terminal is needed to reduce the phrase to <statement> . In some cases we could stop at <specific statement> only. But then for the production <else statement> → ELSE ; , < ϕ* ELSE >* together with any terminal will fail to find an entry in transition matrix. Thus in both the cases error exit would be normally taken. Instead, we branch to the 7th group where we check the U* on stack. If the U* is corresponding to the above cases, the reduction, <statement > → U*, is made and appropriate semantic routine called. If U* is not of the type described above, the incoming terminal is unacceptable and the sentence being recognized is in error.

The two other groups have their origin in the decision to retain the chains of the type $U_1 \rightarrow U_2 \rightarrow U_3$, in the syntax tree. This is to be discussed next.

5.7 Retention of Chains :

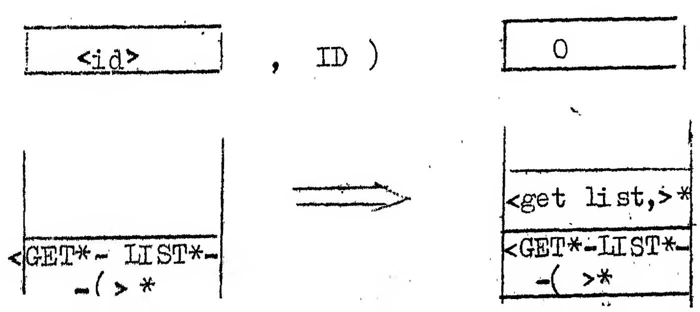
In Patil's constructor no provision is kept for maintaining chains in syntax tree. When a reduction according to condition (2.3) is made ($U \to U_i^* U_k T_j$, and $U_k \overset{*}{\to} U_1$) stack no record is kept of the occurance of U_k.

Thus in the syntax tree the bottom node in a derivation is the only one appearing. This probably is in pursuance of the view of Gries that no interpretation rule is usually associated with $U_k \to U_1 \to \dots \to U_1$. But many times this is not the case.

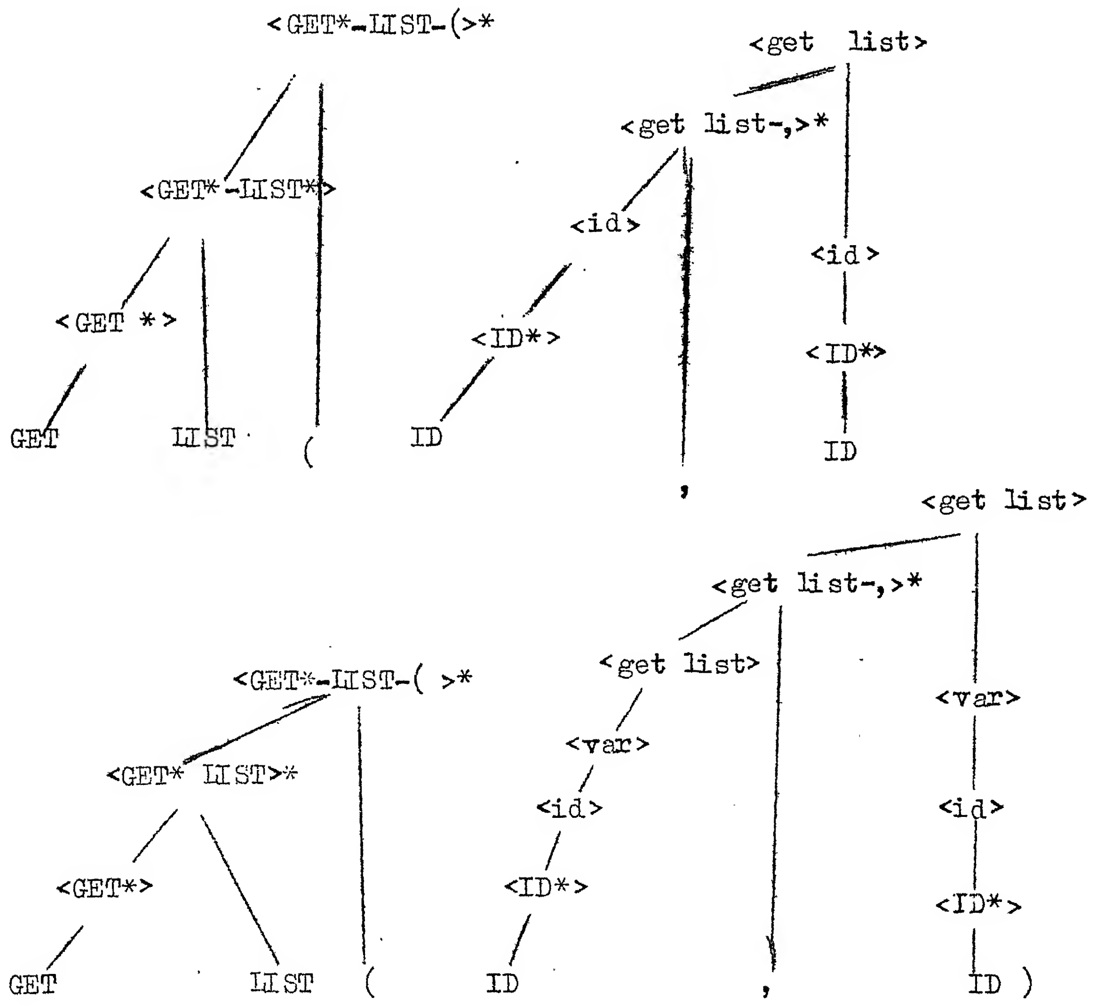
Take for example the following production for I/O statement (Appendix A.2):

<put list stmt>	→	PUT LIST (<put list>)	(1)
<put list>	→	<expr>	(2)
<put list>	→	<put list> , <expr>	(3)
<get list stmt>	→	GET LIST (<get list>)	(4)
<get list>	→	<variable>	(5)
<get list>	→	<get list> , <variable>	(6)

Now in processing a statement GET LIST (ID, ID), we shall have at one stage :



Thus the processing corresponding to get list can be done only when the right bracket has been absorbed. If however, we maintain the chain and duly get the U_k also recorded, We shall come across get list for every ID from rules (5) and (6). The problem between put list and get list does not arise, as the ID's in both cases, first go to a common non-terminal. (fig. 5.8).



: Partial syntax trees with and without chains:

The decision to keep chains involved a modification of the constructor. The description of the modification requires detailed knowledge of constructor mechanism; hence it is not included here.

Ambiguity Resolution : We talked of the ambiguity that is contained in the syntax of expressions in the section 'Syntax-Semantics-Muddle'. The chain maintainance allows the resolution of ambiguity in a convenient way. At present, after the <id> is transformed into <variable>, the <variable> can at times go into different primaries. Take the parsing of the following statement :

A = B ;

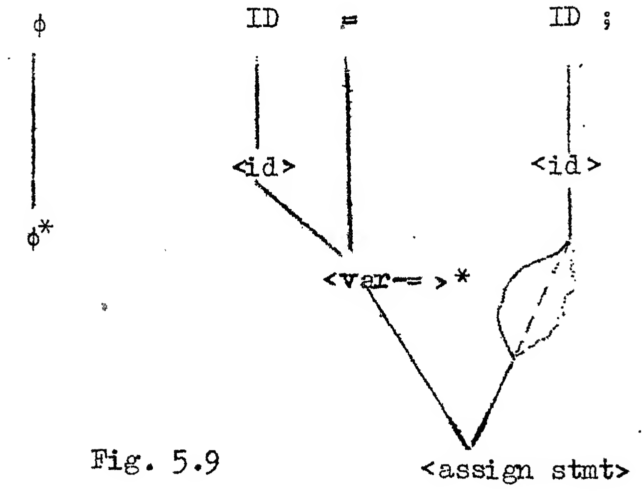


Fig. 5.9

The marked paths are due to the paths via <a-expr>, <b expr> and <char primary> . Thus searching corresponding to B too will have to be done in the <assign stmt> .

Similarly the ambiguities arising (or the searching in symbol table to be done corresponding to <var> → <id> will not

be possible as <var> simply is not formed (it is skipped) in the reduction process. (Note that at <id> no action can be taken because identifiers in many different contexts - labels, procedure names etc. - warrant different treatments). Also care will have to be taken at different points as the above problem may arise when <rel expr> is being formed and in productions where <rel expr> occurs in right parts. This will not only necessitate the use of flags to indicate whether or not symbol table search etc. has been performed, it will also violate the natural and uniform processing by delegating more and more things to the algorithm instead of the tables.

In the modified algorithm when such a situation arises the number of another, i.e. the 9th, group of statements is stored in transition matrix. Now in a situation like (fig.5.10),

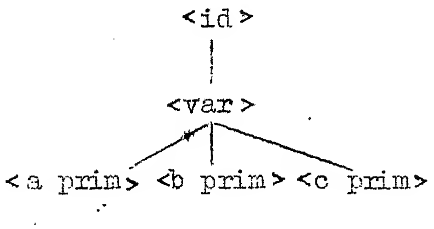


Fig. 5.10

the type of variable is checked and appropriate reduction made. This is much more uniform and conceptionally simpler than handling things at a thousand different places.

While at it, the reason for providing a second syntactic level above <id> , i.e. <id form> , <id proc> , <label> , <var>

etc. may be mentioned. These entities require different treatment which is conveniently provided when <id> is reduced to one of these syntactic entities depending on the context.

5.8 Conclusions:

One of the limitations is the problem in shaping the grammer to meet the conditions 1 and 2, which has already been illustrated by examples. This problem becomes specially important when grammer is extended to include extra features that it does not already have. The people extending it, have to see the implications of extension on rest of the grammer, even when it is obvious that the extensions do not make the grammer ambiguous. Second problem is that of the inordinate size which can be tackled only by using the packing scheme suggested in section 5.3.

One of the major advantages of transition matrix technique is the possibility of giving specific and extensive error messages. This can be done quite easily (manually) by putting the error message number in the Gries' 2dimensional matrix. It seems,however, that the error message should depend upon the U's too and the analysis will have to be done by a subroutine whose number may be put in the Gries' matrix. The process will of course not be so easy in the present scheme. On failing to get the entry in the transition matrix,a second set of tables will

will have to be searched to get the error message number. Manual insertion is not possible as all the terminals (T_j) for a U* have to be together, and insertions thus will spoil the catalogue system. This could, however, be automated by providing the set of error message numbers with the combinations to the constructor, and let it take care of the inclusion of these in transition matrix.

CHAPTER 6

SYNTAX ANALYSIS AND PROGRAM STRUCTURE CHECKING

In chapter 4 we gave a general description of the whole implementation. The intent of this chapter is to go into the details of the syntax analyzer as well as the driving routine and also talk a little about the general organisation of the main semantic routine. The details of individual semantic tasks will be given in chapters 7 (Part I and II). The description of semantics in the present chapter will be confined to the checking of program structure and the initialisations etc., required for processing new statements or procedures. Most of the description will be based on flowcharts. Discussion of special strategies etc., will be given when warranted. For details of common areas and description of various data refer to Appendix

6.1. The Driving Routine :

The main purpose of the driving routine ANDRIV is to look after the transition from job to job as well as transferring control to the quadruple processor. Details of how this control is exercised is described in section 4.6. At present we shall assume that the task includes compilation of single job.

The block diagram (fig. 6.1) is self-explanatory. The only thing that need be explained is as to what initializations are

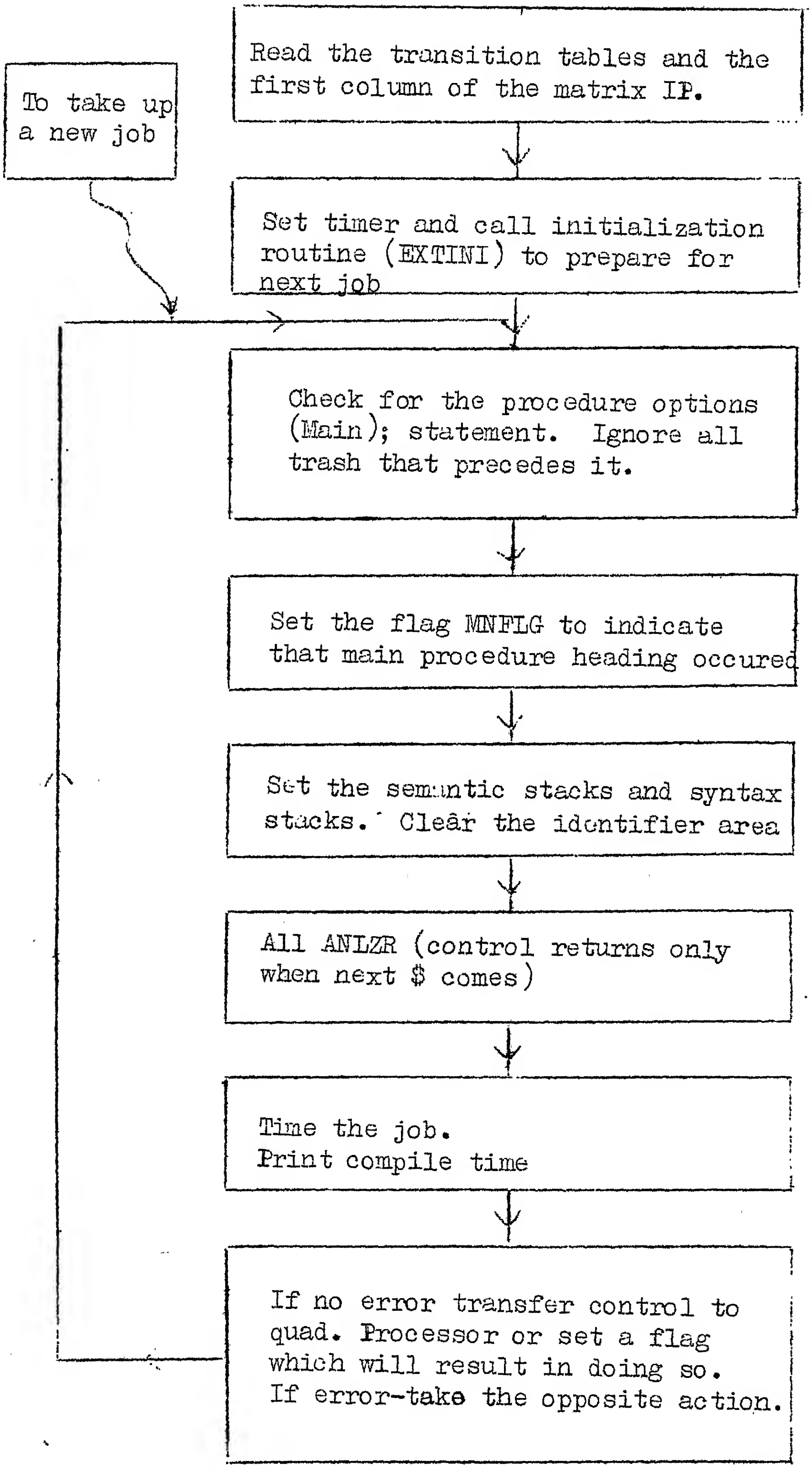

Read the transition tables and the first column of the matrix IP.
To take up a new job
Set timer and call initialization routine (EXTINI) to prepare for next job
Check for the procedure options (Main); statement. Ignore all trash that precedes it.
Set the flag MNFLG to indicate that main procedure heading occured
Set the semantic stacks and syntax stacks. Clear the identifier area
All ANLZR (control returns only when next $ comes)
Time the job.
Print compile time
If no error transfer control to quad. Processor or set a flag which will result in doing so.
If error-take the opposite action.

Fig. 6.1

done,where. There are two main initialisation routines EXTINI and INIANL. Extini initializes various pointers and semantic stacks and tables etc. The iniatialization of individual stacks etc., is given in the proper context, i.e. at the places they are used. INIANL is a routine that initializes the syntax stack and this initialization too is described in the description of syntax analyser that follows.

6.2. The Syntax Analysis Routine :

The routine ANLZR is activated by the driving routine ANDRIV and it transfers control back to ANLZR only when the character $ signalling the end of job is encountered.

Primarily the syntax analysis process goes by using the recognition technique described in chapter 5. A push-down stack STAK1 is used to hold starred nonterminals and a variable NL to hold the current (if any) unstarred nonterminal (U). The stack STAK2 associated with STAK1 was earlier used for holding the ranges of U*s temporarily. TREE1, TREE2 and TREE3 are three stacks used to hold the partial syntax trees and associated semantic information. In the initial version a linearized form of the tree where pointers with the nonterminals to hold the last terminal in the range were kept (section 5.6) was used. In the newer version the production numbers are loaded on stack and there is no necessity to keep the range. As such the statements relating to the range are superfluous but have not been removed to permit later modification, if any.

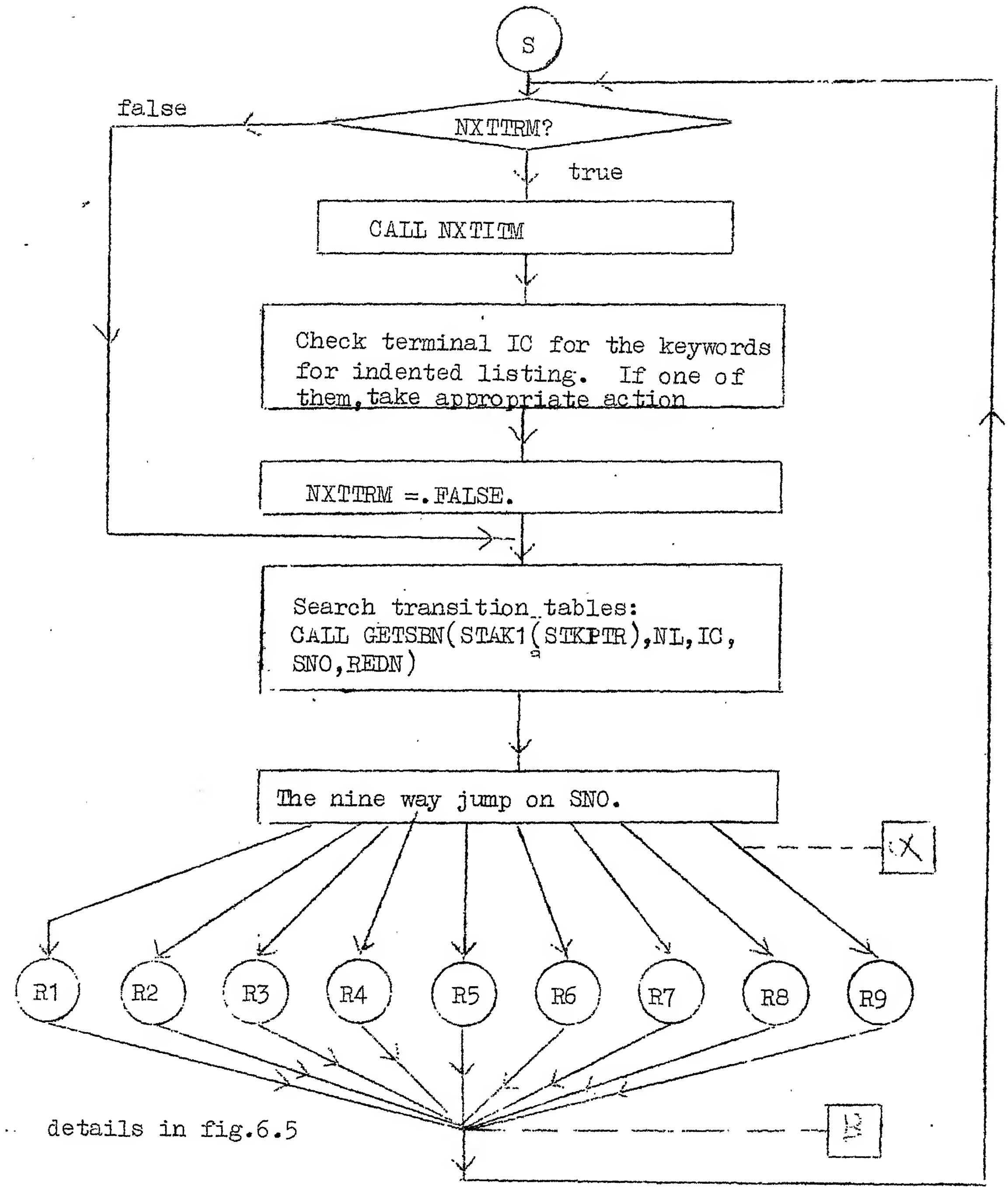


Fig. 6.2

Subrontine ANLZR:

The recognition process (fig. 6.2) goes as follows :

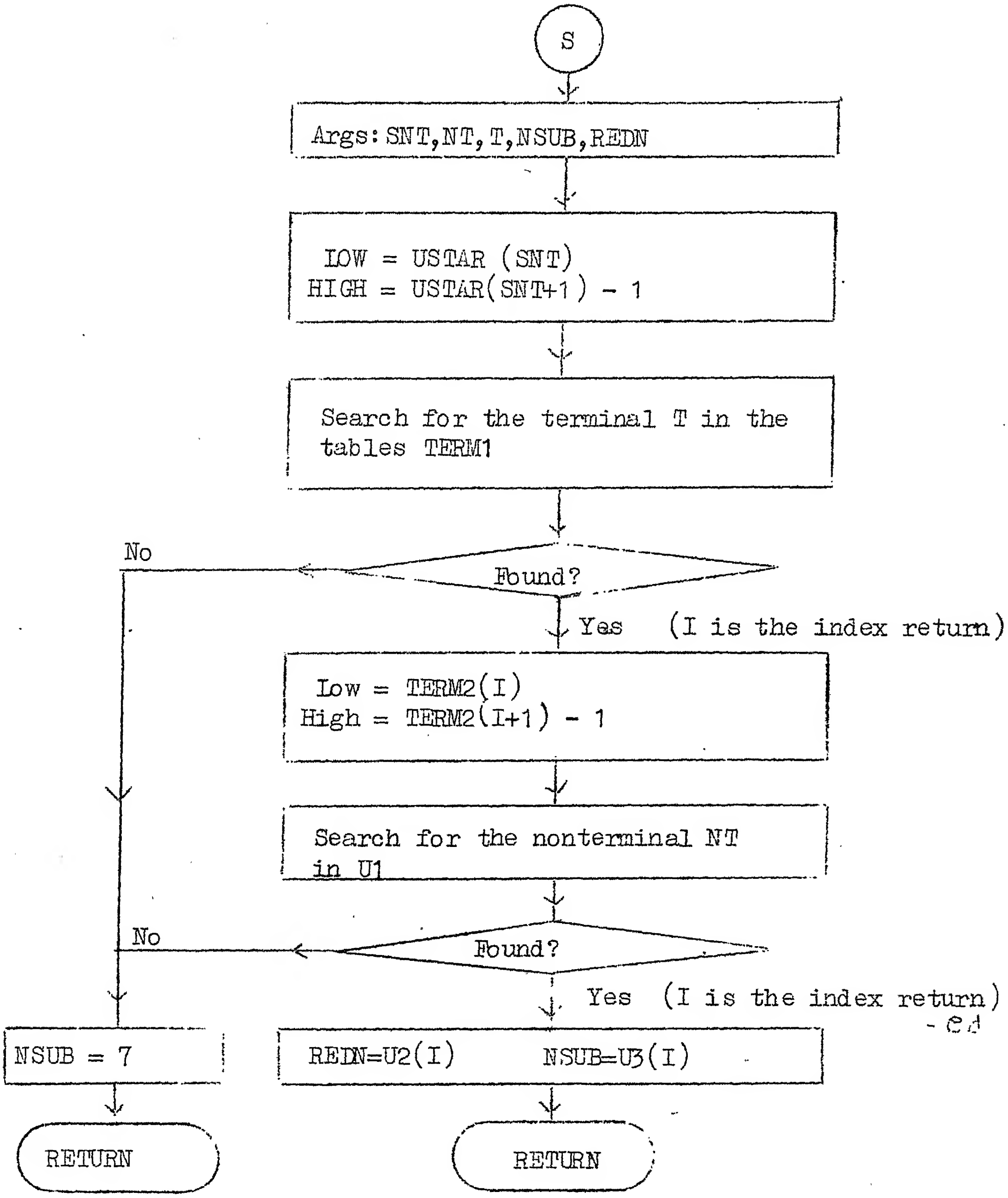
a. Initialization : Syntax stacks are initialized in routine INIANL called once before ANLZR is activated. Initialization consists of setting pointers for STAK's and TREE's to 1 and putting the codes for ϕ* in STAK1 and ϕ in TREE1. ϕ* is a special starred non-terminal which derives the special terminal ϕ which is assumed to be appended to every sentence to be recognized. NL is set to zero.NXTTRM which a flag indicating whether a new terminal is needed (true) or not (false) is set to TRUE.

b. Getting the Next Lexical Unit :

The lexical processor NXTITM is called to get the code for the next terminal (Lexical unit). The information is placed in the common area LEXITM. IC is the code of the terminal returned. IC2 is secondary information, e.g. the type (FIXED, FLOAT, etc.) associated with the lexical unit TYPE or pointers to constant tables in case of constants. The code is examined to take actions required to properly indent the output listing. For details of indentation scheme refer to section 6.3. If the code is of $ then control returns to the driving routine ANDRIV. NXTTRM is set to false. It is reset to true if, depending upon the reduction to be made, a new terminal is required.

c. Getting the type of reduction to be made :

Subroutine GETSBN (fig. 6.3) is called with arguments : the unstarred nonterminal (U*) on top of stack (STAK1 (STKPTR)),



SNT: Starred nonterminal NT: nonterminal T: Terminal

NSUB: The type of reduction. TRANS is the common area for transition tables REDN: The code for the starred nt. or the prodn. no. given in the array U3.

:Subroutine GETSBN: Fig. 6.3

the unstarred nonterminal (NL) and the present terminal IC. The subroutine GETSBN searches the transition tables and returns one of the eight (1-6 and 8,9) type of reductions to be made (SNO) and the code (REDN) of the left hand side of the production to be used (code for U* or the production number (in case of a nonterminal U)). When GETSBN does not find an entry it returns 7 as the value of SNO. SNO is used as an index to a computed GO TO to transfer control to one of the 9 groups of statements for making different reductions.

d. <u>The Reductions</u> : (fig. 6.5)

The reductions R1 to R6 are corresponding to the six conditions given in chapter 5. The significant points about these are :

After taking care of the STAK1 and NL (found from the array IP of the nonterminals for different rules), the routine SEMAN is called if the reduction is of the type U → U*U or U → U*, the argument being the production number. The routine ASTREE is called to build the syntax tree. The flag NXTTRM is set to <u>true</u> if the reduction uses the terminal.

The reductions R8 and R9 have been introduced in the present implementation to build the chains in syntax tree described in section (5.8). R8 calls the routine SEMAN and then calls ASTREE to build the chain in TREE1. R9 is to take care of the case-where ambiguity arises,as described in chapter 5. An examination of the

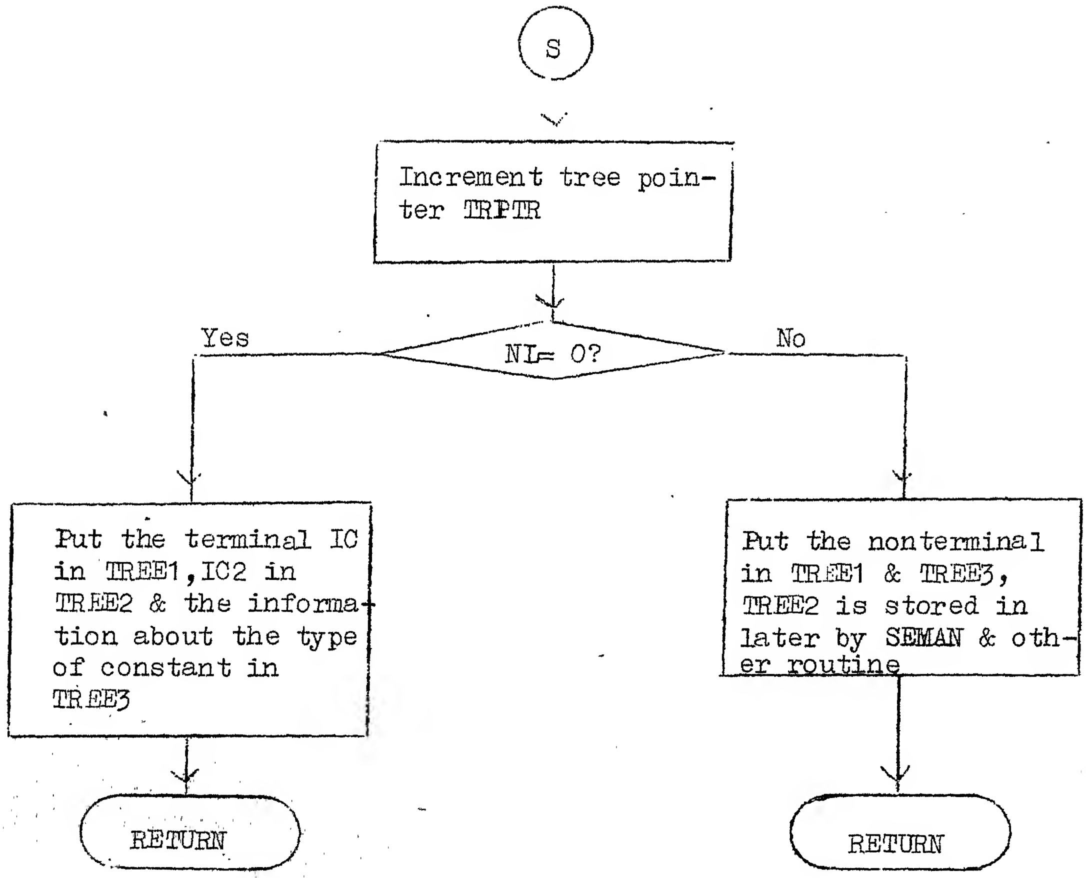
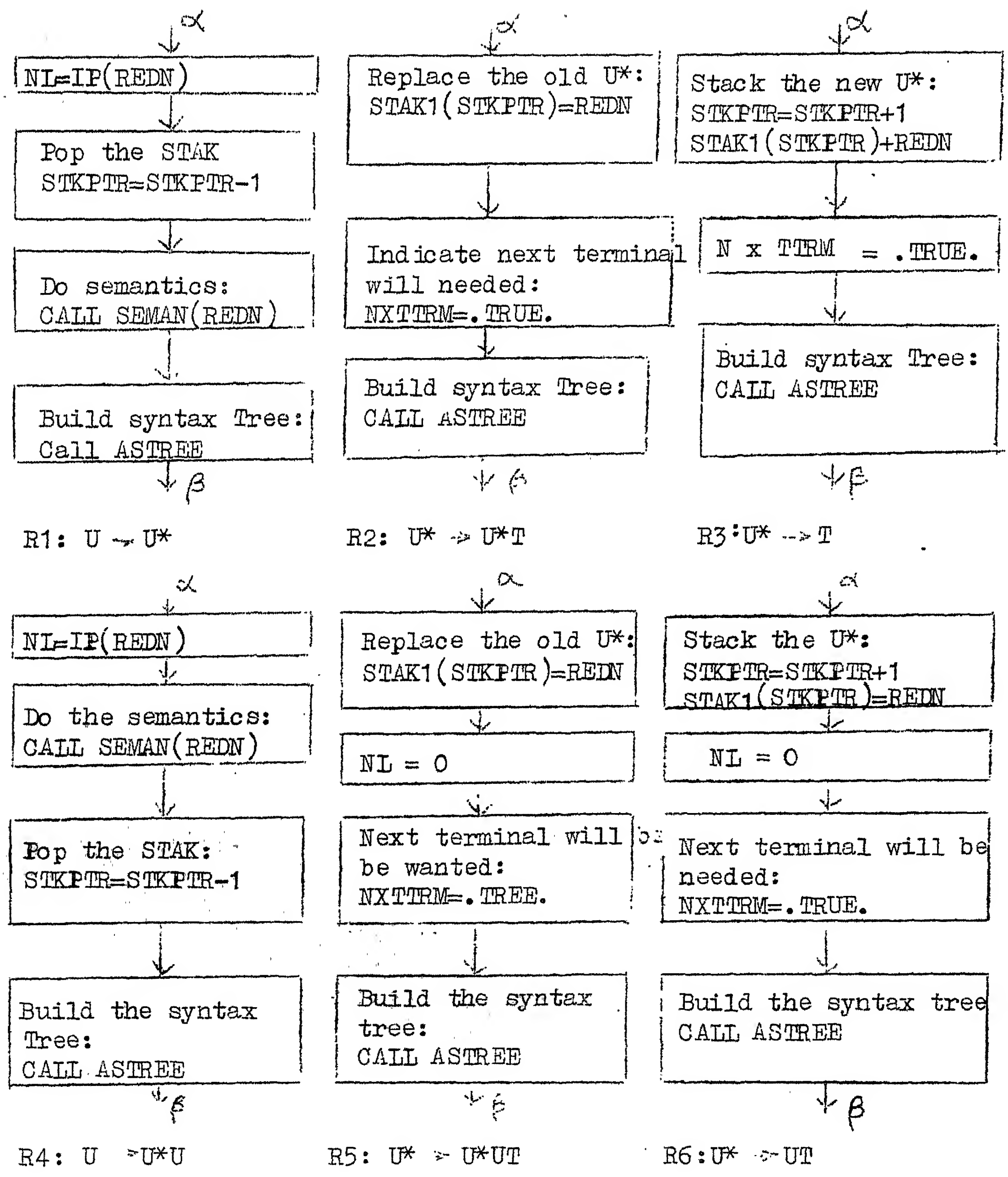


Fig. 6.4

ANALYS is the common area holding stacks STAK & TREE

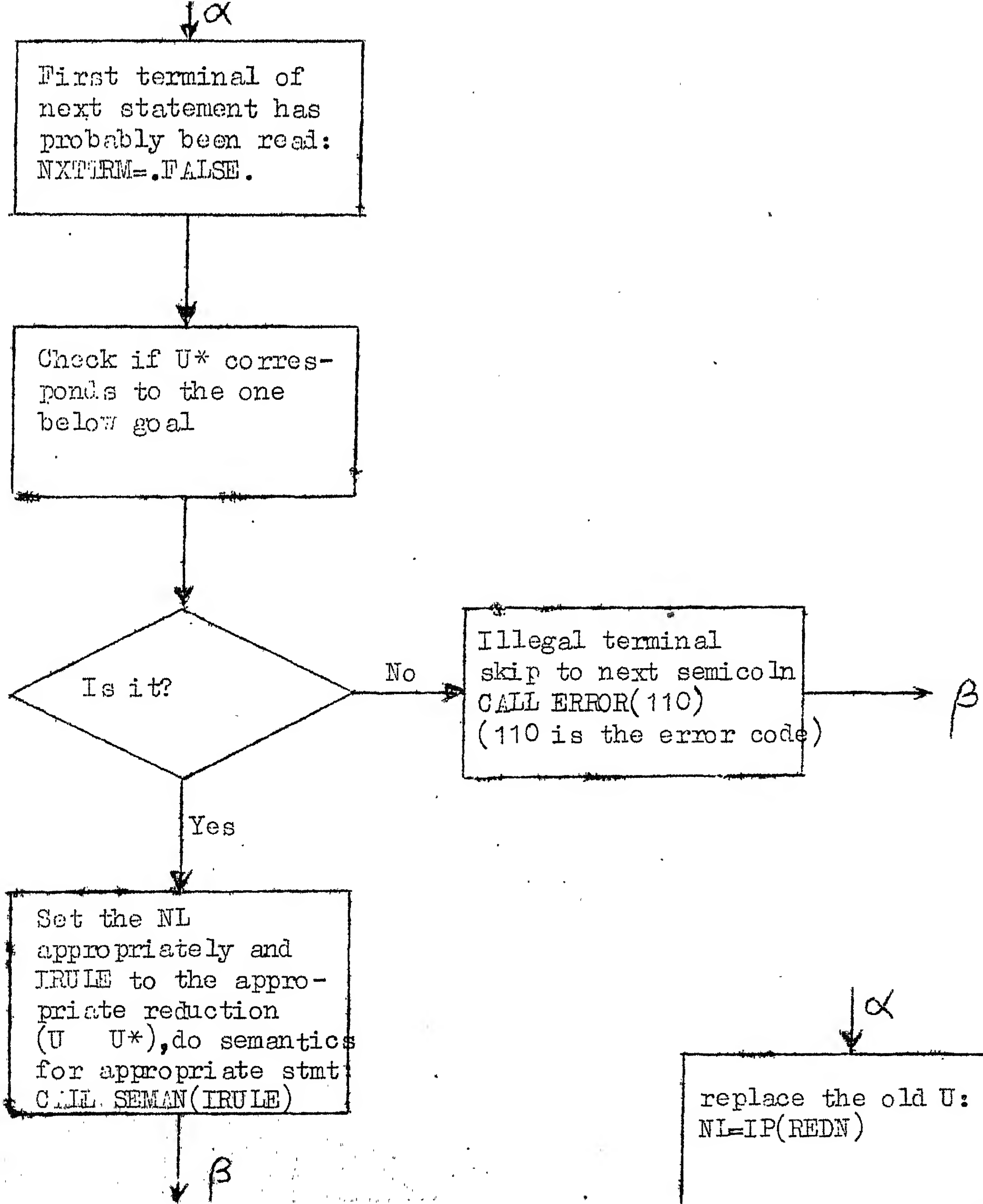
LEXITM is the common area holding IC & IC2 etc.

Subroutine ASTREE:

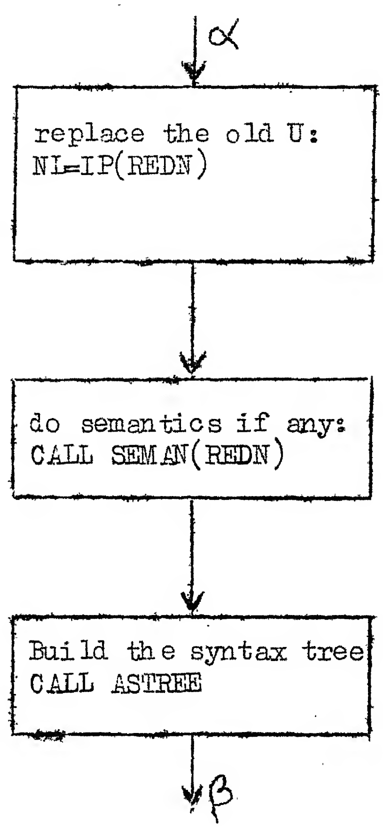


The nine groups of statements for different reductions:

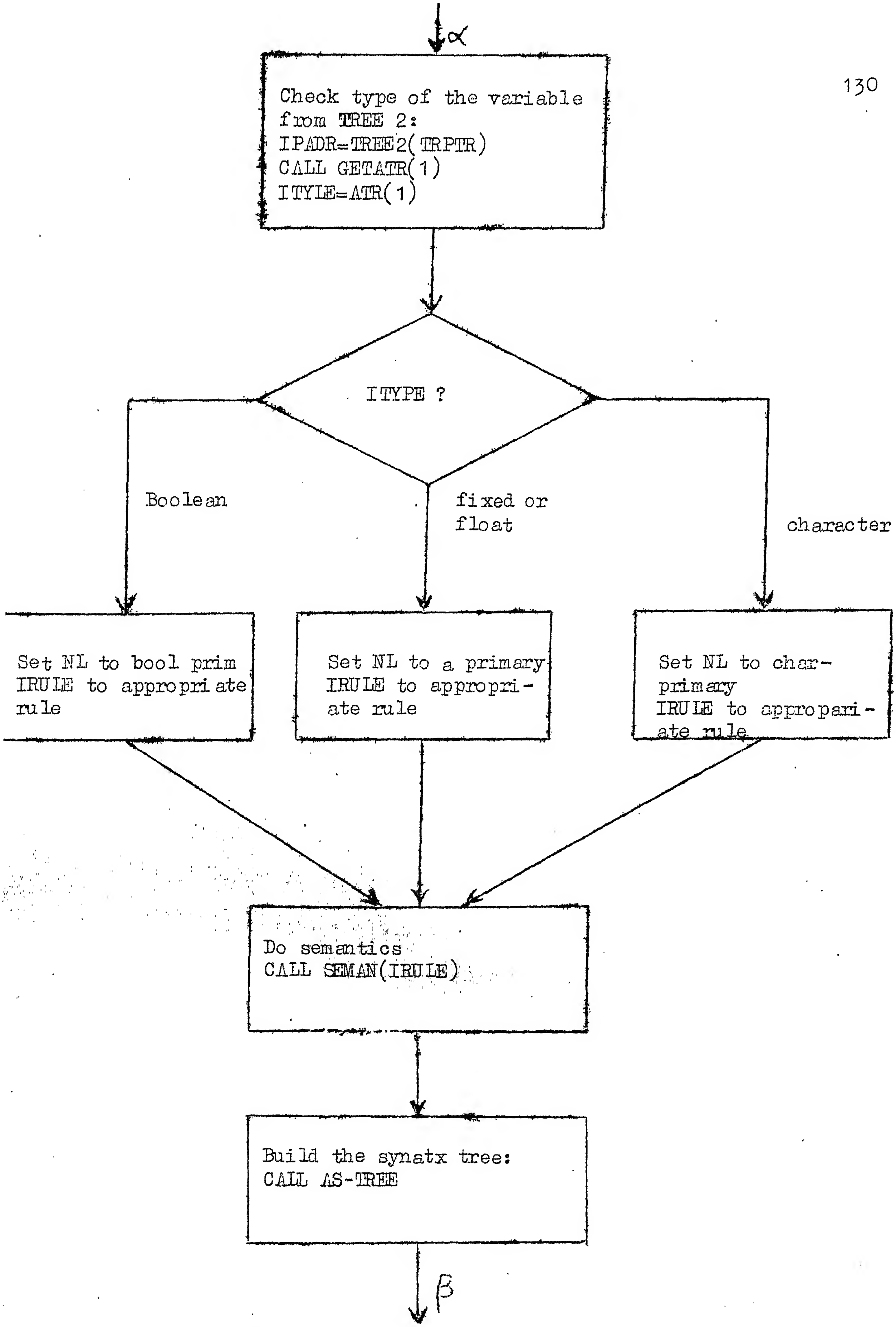
Fig. 6.5a



R7: No entry in transition table.



R8: $U1 \rightarrow U2$



R9: Resolution of Ambiguity

type of variable is made and proper reduction decided accordingly.

Reduction R7 is to take care of the recognition of the final goal or nonterminals 52, 53, 54 . As pointed out in chapter 5 at the end of reduction of a statement, a U* corresponding to these goals keeps sitting on the STAK1 and a reduction of the form U→U* is required. When the next terminal is absorbed an error condition occurs and GETSBN returns the SNO as 7. Then R7 checks for the presence of proper U*'s on the stack. If found, proper goal nonterminal is set as NL and the routine SEMAN called. If not the incoming nonterminal is in error and the subroutine ERROR is called with appropriate error code. ERROR routine in this case skips to the next semicolon and calls INIANL to initialize the syntax stacks etc. INIANL is also called by SEMAN when a statement is recognized.

6.3. Formatting the Program Listing :

The above feature referred to in chapter 4, though doubtless a great aid to program readability, has been incorporated quite simply in the fashion described below :

An output buffer of 131 characters is maintained. The routines GETCHR and NOBLNK of lexical keep putting characters in it. Print-out takes place by explicit commands to the routine LISTOU. This occurs, when (1) output buffer becomes full in GETCHR or NOBLNK routines (2) a ';' , THEN or ELSE is supplied to ANLZR, (3) a comment occurs (this is printed on a new line).

To properly indent output listing the margin is increased or decreased by signalling to the routine LISTOU by raising in the routine ANLZR the flags INCFLG and DECFLG respectively. IF-THEN, DO, BEGIN, PROCEDURE increase the margin. ELSE and END restore the margin. In case of errors sometimes the part of the statement in error gets printed out followed by the error message and then by the remaining portion of the statement. A sample of the unformatted input program and the formatted program listing is given in Appendix C.

6.4. Checking the Program Structure :

The goal nonterminal of the syntax analyzer could have been kept as the <program> itself. <statement> could also have been defined as, <simple statement> and <compound statement> . Thus by making various compound statements and procedure etc., as syntactic entities, the structure of the program could have been checked in the syntax analysis process itself. This has the greatest disadvantage of exploding the transition matrix size, which is already critical, possibly to unaccomadable volume. Also PL/I (and thereby MINIPL)has a syntax quite naturally fitting into a grammer where the smaller units are treated as goals. The explicit termination of statements by';' is in contrast with Algol where the compound statement is absolutely identical to a simple statement, in as much as this too is terminated by the semicolon. However, treating the simple statements, as goals necessitates maintainance of history in

another stack as will be seen in the following material.

The productions with L.H.S. as the nonterminals 52, 53, 54 (Appendix A.2) are corresponding to the definitions of statements. When SEMAN is called with these production numbers as the argument, processing to check the properiety of the program structure is carried out.

The description in this section mainly concerns itself with the portion (not necessarily physically) of SEMAN that deals with the structure checking. The approach will be mainly based upon the semantics stacks and their use to check structure. However, the semantics, specially generating jumps etc. for conditional statements is deeply related to the matching and nesting of clauses and these are best described together. Thus in the present section we shall describe the semantics for the aforesaid statements; for the other ones details of semantics will be given independently. First we shall look at what is involved in structure checking, then give the method adopted followed up by a description of the main semantic routine SEMAN to fix up the ideas.

For the purposes of structure checking we shall classify statement into the following categories :

1. <if-then stmt> 2. <else stmt> 3. <begin stmt> 4. <Do stmt>
5. <end stmt> 6. <Declare stmt> 7. all other <stmt>s .

Things to be checked are :

a. Nesting of DO - groups and PROCEDURE and BEGIN blocks.

b. Proper nesting of the IF-THEN and ELSE statements

c. Occurances of declarations and procedure definitions at the proper place in any block, i.e., declarations first, then procedures followed by executable statements.

As for the check (a), all that is required, is to push the code for do, begin, or procedure onto the semantic stack. Also a flag is maintained for each block which tells whether an executable statement a declaration or a procedure definition has occurred in the block or after the most recent if-then or else statement , so far. To see what is involved in checking (b) let us examine the if-then-else statement of MINIPL.

The general format is,

$$\underbrace{\text{IF b exp THEN S1}}_{\text{if-then statement}}\ ;\ S';$$

$$\underbrace{\text{IF b exp THEN S1; }\underbrace{\text{ELSE S2;}}_{\text{else -stmt}}}_{\text{if-then-else-stmt}}\ S';$$

where, S1 and S2 are either simple or compound statements. The occurance of the statement S' indicates the completion of the previous if-then or the else-stmt . An if-then stmt immediately followed by an else-stmt forms an if-then-else stmt . The problem is to see if these conditional statements are proper becomes complicated in the situation where there is a nesting of conditionals.

Example :

IF <b exp1> THEN IF <b exp2> THEN <stmt1> ; ELSE

IF <b exp3> THEN <stmt2> ; <stmt3> ;

Now, the fact that the first if-then does not have an else counterpart can be knownonly when stmt3 is recognized. The most important thing in the semantics of if-then-else statements is provision of proper jumps.

Example :

a. IF <b exp> THEN <sl> ;

When the <b exp> has been evaluated a system label is generated and a conditional go-to to this statement is generated. After sl is recognised the old label generated must be defined.

b. IF <b exp> THEN IF <b exp'> THEN <s1> ;

The label defined for the first if-then statement can be used for the go-to in the second if-then too as in case of any of the <b exp> 's being false a jump beyond the <s1> is to be executed. This label will be defined after the statement <s1> is recognized. The structure processing part of the semantics assumes that rest of the semantics for the statements are already over (e.g. evaluation of <b exp> in case of an if-then statement).

The Structure Checking Algorithm :

The main stack for semantics is in fact a set of stacks MSTK1, MSTK2, MSTK3, which store information about the type of

currently active program structure entity, the flag indicating the status within the structural entity (whether a declaration, procedure definition or an executable statement occurred), and names of associated system generated labels, if any. The details of label handling are found in chapter 7, at present We shall just give the operations we want on them, here.

The algorithm described below gives the actions taken for each of the 7 categories of statements defined on page

Initialization : It is assumed that an external procedure is sitting in the stack.

Step 1. Place the top elements of the three stacks MSTK1, MSTK2, MSTK3 in the variables TOP, STATOC and LABOLD (changes in STATOC etc., will indicate that top of stack is being changed too , popping subsequently will include the appropriate adjustment of the three variables).

Step 2. Jump to appropriate step depending upon into which of the seven categories referred to above, the present statement falls.

Step 3. <if-then stmt>

3a. If TOP is not a conditional, then make the STATOC = 1, push the current if-then on MSTK1 with the MSTK2 as 0 (no statement yet occurred, generate a new label and push it on MSTK3, and issue a conditional go-to to this label, return. If TOP indicates conditional, go-to step 3b.

3b. If no statement had occurred (i.e. STATOC = 0) after the previous conditional, push the <u>if-then</u> on MSTK1, generate a new label and push it into MSTK3, issue a conditional <u>go-to</u> to this label, <u>return</u>. If a statement had occurred (i.e. STATOC ≠ 0) then go to step 3c.

3c. Pop the TOP until it indicates something other than an <u>if-then</u> or an <u>else</u>, push the new <u>if-then</u> onto MSTK1, generate a new label and issue a conditional <u>go-to</u> to it, <u>return</u>.

<u>Step</u> 4. else stmt

4a. If TOP is not a conditional, then print error, <u>return</u>. If it is, go to step 4b.

4b. If no statement occurred before this <u>else</u> (STATOC = 0), print error, <u>return</u>. If one did occur go to step 4c.

4c. If TOP is an <u>if-then</u> , pop the <u>if-then</u>, push the <u>else</u>, generate a new label, issue an unconditional <u>go-to</u> to it, define the LABOLD, push the new label onto MSTK2, <u>return</u>. If TOP was not <u>if-then</u>, go to step 4d.

4d. (The TOP already has an <u>else</u> which had a statement after it). Pop the <u>else</u> on TOP. Go to step 4a (to find a matching <u>if-then</u>).

<u>Step</u> 5. <begin statement>

push the begin on TOP with STATOC = 0; do rest of the semantic processing, <u>return</u>.

<u>Step</u> 6. <do statement>

push the do on TOP, push the label for <u>return</u> from <u>end</u> in MSTK2 (STATOC not used for do), generate a label, push it on MSTK3 and issue a conditional <u>go-to</u> to it (this condition is the termination condition of the iterative <u>do-group</u> which has already been evaluated. If non-iterative <u>do</u>, this action is not to be taken).

<u>Step</u> 7. <end statement>

7a. If TOP is not one of <u>begin</u>, <u>do</u> or <u>procedure</u>, print error, <u>return</u>. If it is, then go to step 7b.

7b. If TOP is <u>begin</u> or <u>procedure</u>, do the required semantic processing. If it is <u>do</u>, then issue a <u>go-to</u> to the label for another iteration, define the label for the termination <u>go-to</u>. Pop the TOP. If TOP was procedure, <u>return</u>. If TOP was <u>begin</u> or <u>do</u>, go to step 8. (Since the compound statement is now recognized, which is equivalent to an ordinary statement for purposes of <u>if-then-else</u>.)

<u>Step</u> 8. Other <statement>s

8a. If a statement did not occur, make STATOC = 1 (statement occurred). If IC (next terminal) is an <u>else</u>, <u>return</u>. If it is not, go to step 8b.

8b. If a non-conditional on TOP, <u>return</u>. If a conditional on TOP go to 8c.

8c. Define the LABOLD. Pop the stack, go to 8b.

<u>Note</u> : A point or two about the above algorithm are in order. The processing for the begin and procedure blocks and their physical ending involves tasks related to storage allocation and symbol table management etc., and hence it's description is included in chapter 7. Secondly, there is a slight difference in the order of actual processing for the category 'other statements'. The main semantic routine SEMAN first does the processing related to structure and then goes to the 'main computed go to' for processing of individual statements.

The following example will illustrate the algorithm described above :

```
                                          Code generated
BEGIN^1
 :
IF <b exp>^1 THEN^1                       Branch on false, <b exp>^1 , LAB1
     IF <b exp>^2 THEN^2                  Branch on false, <b exp>^2 , LAB2
                                            :
         BEGIN^2;                           :Code for Begin^2 block
           :                                :
         END^2;                             :
     ELSE^2                               Branch, LAB3
                                          Define, LAB2
         IF  b exp ^3 THEN^3              Branch on false, <b exp>^3, LAB4
         S2 ;                               :Code for S2
                                             Branch, LAB5
         ELSE^3                           Define, LAB4
         S3 ;                               :Code for  S2
                                             Define LAB5, LAB3, LAB1
         S4 ;                               :Code  for S4
 :
 :
END^2
```

Note: The superscripts in the preceding program are just for clarity. OCCUR indicates a statement has already occurred; $\overline{\text{OCCUR}}$ indicates, it has not. b exp in the following figure, will represent the location in which the result of evaluating b exp is stored.

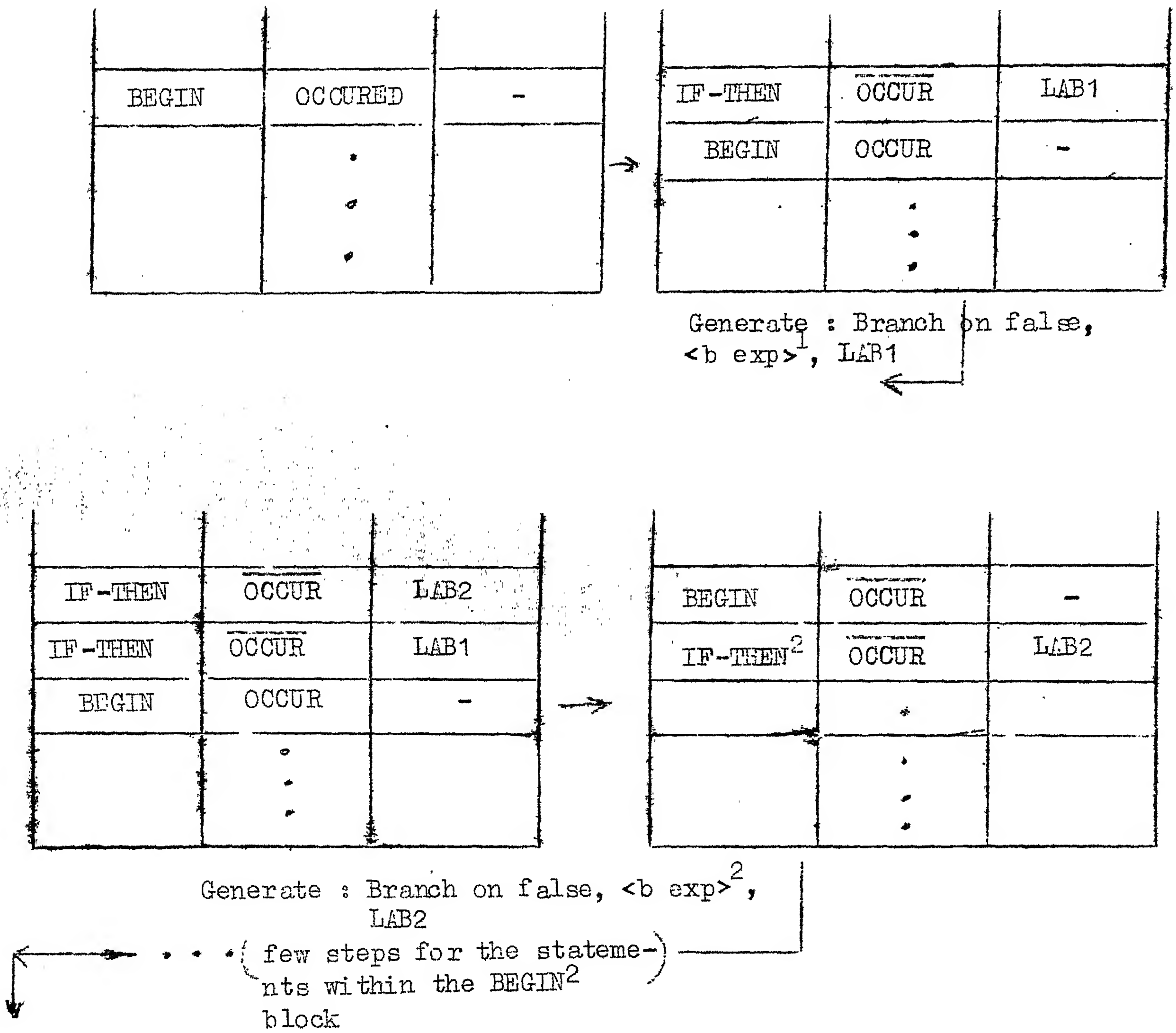


Fig. 6.6a

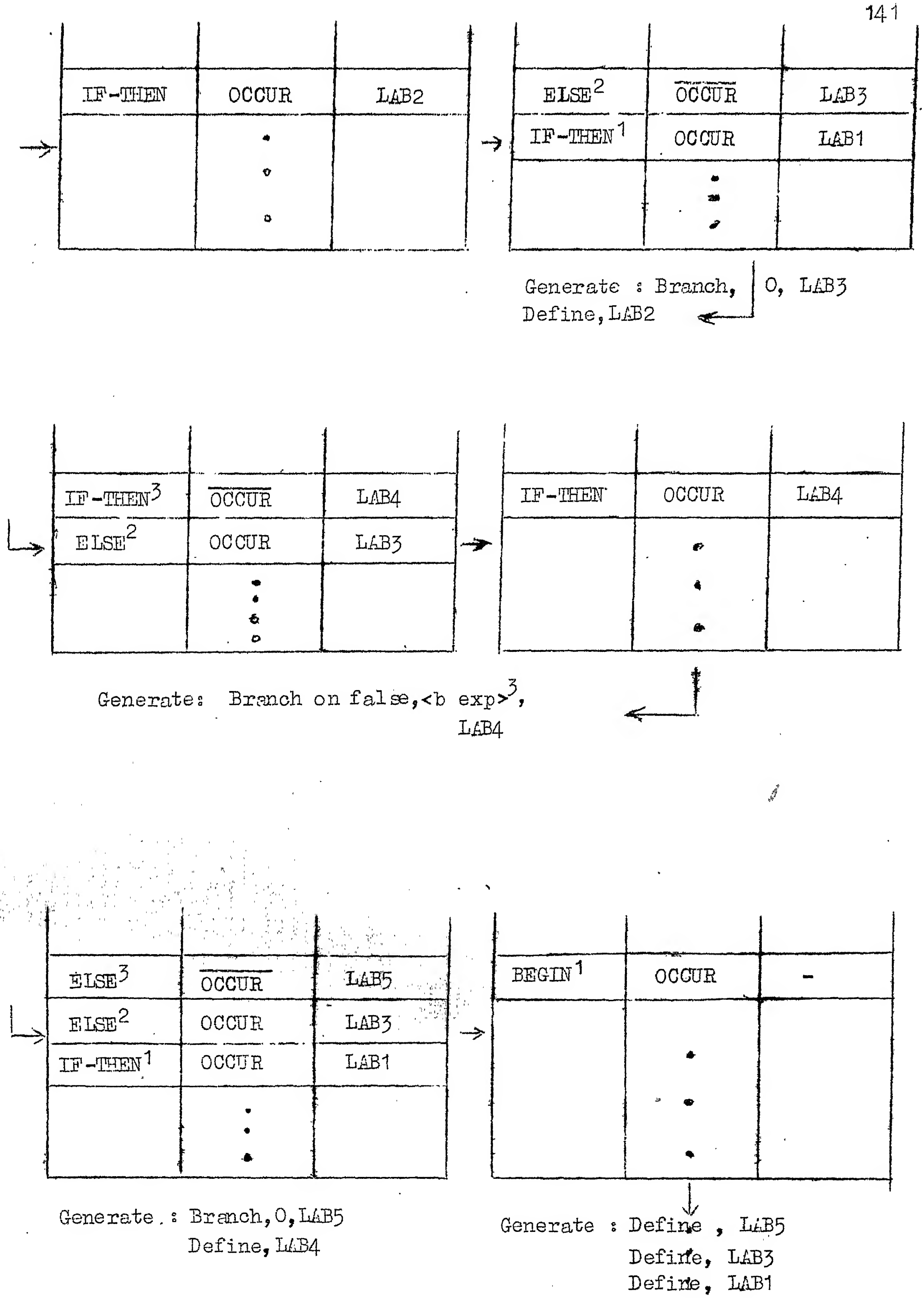

IF-THEN
OCCUR
LAB2
ELSE²
OCCUR
LAB3
IF-THEN¹
OCCUR
LAB1
Generate : Branch, 0, LAB3
Define, LAB2
IF-THEN³
OCCUR
LAB4
ELSE²
OCCUR
LAB3
IF-THEN
OCCUR
LAB4
Generate: Branch on false, <b exp>³,
LAB4
ELSE³
OCCUR
LAB5
ELSE²
OCCUR
LAB3
IF-THEN¹
OCCUR
LAB1
BEGIN¹
OCCUR
-
Generate : Branch, 0, LAB5
Define, LAB4
Generate : Define , LAB5
Define, LAB3
Define, LAB1

Fig. 6.6b

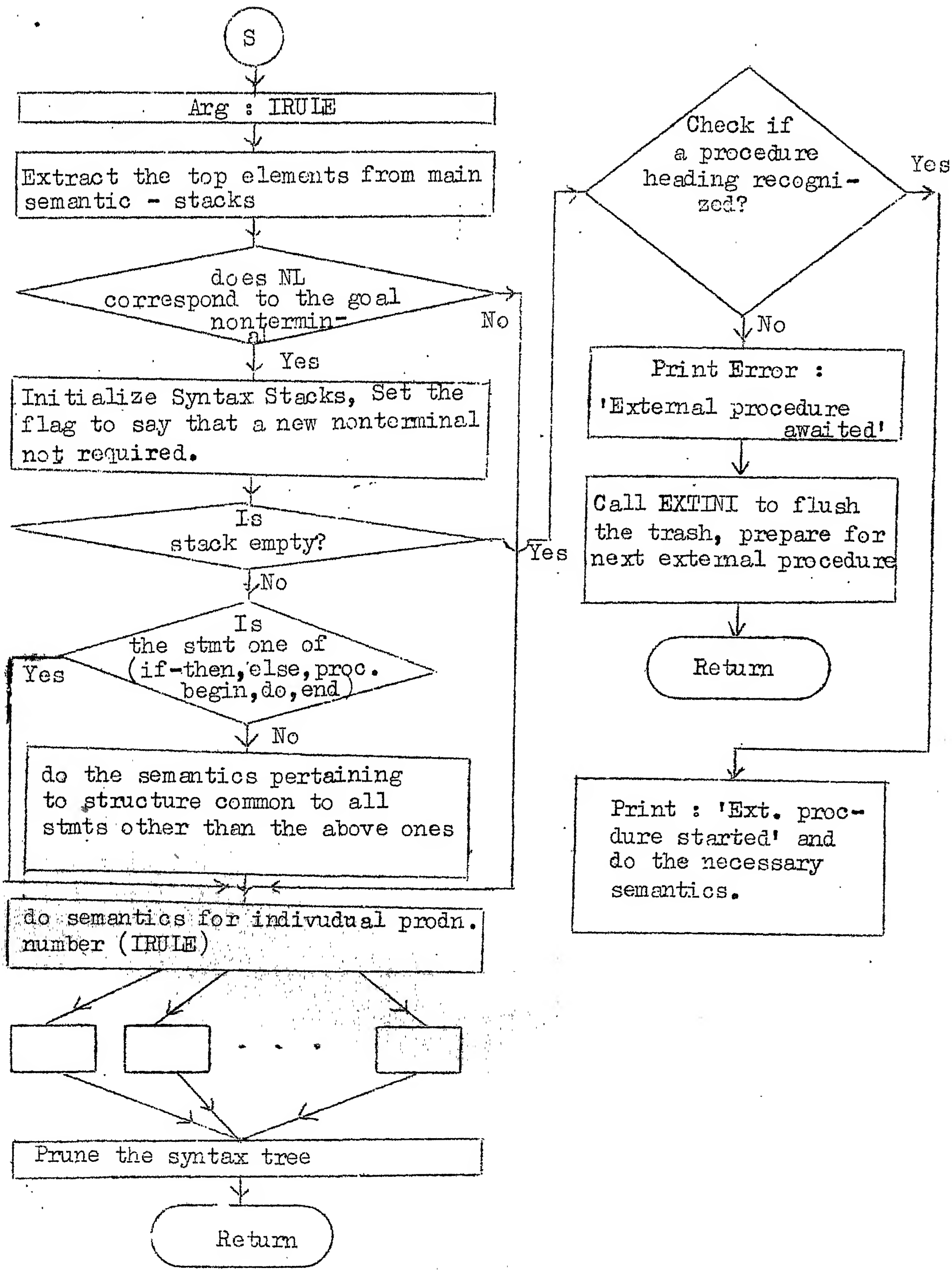

S
Arg : IRULE
Extract the top elements from main semantic - stacks
does NL correspond to the goal nonterminal
No
Yes
Initialize Syntax Stacks, Set the flag to say that a new nonterminal not required.
Is stack empty?
Yes
No
Is the stmt one of (if-then, else, proc. begin, do, end)
Yes
No
do the semantics pertaining to structure common to all stmts other than the above ones
do semantics for individual prodn. number (IRULE)
Prune the syntax tree
Return
Check if a procedure heading recognized?
Yes
No
Print Error : 'External procedure awaited'
Call EXTINI to flush the trash, prepare for next external procedure
Return
Print : 'Ext. procedure started' and do the necessary semantics.

Fig. 6.7

6.5 The Routine 'SEMAN'

SEMAN is the main semantic routine and is activated from syntax analysis routine ANLZR with production number (IRULE) as the argument. It does the semantic processing for the syntactic entity recognized and stores information in the secondary syntax stacks (TREE2 and TREE3) and also maintains main semantic stacks described in the previous section, for structure check and other reasons. Due to the large variability of the information maintained and manipulated for semantic interpretation of different syntactic entities, it will be neither meaningful nor feasible to describe the whole program here. The flow chart in fig. 6.7 gives at a macro-level the general flow of control valid for all syntactic entities as a whole. An important array not mentioned before is RHS. This contains the number of terminals or non-terminals in the right hand side of each production of the input operator grammer. As shown in the flow chart, TREE1 is popped to clean all the symbols in the present handle. If however the symbols have to be retained (though it is dangerous and the effect should be taken care of) the corresponding R.H.S. entry can be made zero. Notice that information attached about the handle in TREE2 and TREE3 becomes associated with it when the new non-terminal (to which the handle has been reduced) is pushed into the syntax tree TREE1 by the routine ASTREE.

CHAPTER 7

SYMBOL TABLE MANAGEMENT, STORAGE ALLOCATION AND RUN TIME ADDRESSING

7.0 In this chapter we discuss three important aspect of the MINIPL compiler sementics. Various basic problems,alternative solutions, design decisions for the present compiler and essential details of the present implementation are given. In section 7.1 we discuss the symbol table organization and the processing of declarations. The next section describes allocation of addresses to variables in MINIPL. The section also outlines the features of a Quadruple Processor which will make effective use of the work done at compile time, by contrasting it with the implementation utilizing the MAP assembler. In the last section (7.3) we describe the run time storage administration and also the relationship between the instruction set of a particular machine (IBM 7044) and dynamic area addressing.

7.1 Symbol Table Management :

Symbol table is one of the most important repositories of information in the compiler. It stores information about all the identifiers-labels, variables, formal perameters, procedure names etc.- of the program. It is accessed by most semantic routines to extract the information about an identifier. Hence it is essential that its organization allow efficient searching and extraction of attributes.

7.1.1 Basic Organization :

Evans (10) has described the symbol table as a means of converting the 'character strings' representing identifiers into integers which can be used as indices to access a table of values, each entry of which corresponds to one entry in symbol table and contains the information for that identifier. We shall consider the symbol table as consisting of an argument column and one or more value (or attribute) columns storing information about the identifier. This description implies that for every identifier in the symbol table same number of cells are kept for storing its attributes, possibly a waste if the attributes for this particular identifier are less than the maximum number possible. Probably keeping this in mind Ramarao (27) has provided for a mechanism of obtaining cells from a free list and linking them up in a list to form the set of attributes for a given identifier. However the mechanism of accessing attributes becomes slow for two reasons: chasing the pointers and seeing the class to which an attribute belongs (this is fixed in the tabular organization by assigning one column to each of the classes). A class in the above context is defined so that an identifier may not have two attribute in the same class (e.g. fixed/float/char/... may form one class, static/automatic/..., forming another). The number of such classes for different identifiers does not vary considerably in a reasonably modest language-making the waste in the tabular organization minimal. This is the case with MINIPL too, and we have a tabular organization,details of which are to be found in section 7.1.3.

If the identifiers are required to be of fixed length the argument column of the symbol table contains the identifier itself. In most cases however, as also in MINIPL, the length of the identifiers is variable. In such a situation, the argument column points to an area where identifier strings are stored. The length of the string may be stored in either the argument column or at the beginning of the string itself. The latter scheme (fig. 7.1) is used in present implementation.

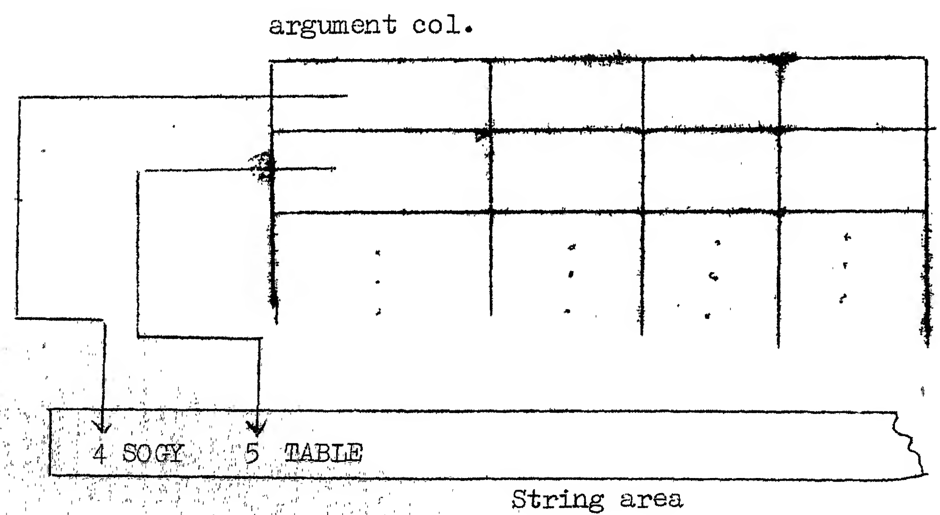


Fig. 7.1

7.1.2 Block Structure and Symbol Table Organization :

Evans' (27) consideration of replacing an identifier, as soon as it is read in, by an integer (index) in the symbol table comes very natural in a language like FORTRAN. There, for the first appearance of an identifier it can be entered into the symbol table, all later appearances being replaced by the index of this entry. With the block structured language-MINIPL is one of them -

the situation is not so simple. The same identifier may be declared and used many times in different blocks and procedures, and each such declaration must have a unique symbol table entry associated with it.

The technique used in ALCOR compiler (17) was to prepare an identifier list and a block list. The latter pointing to the id-list to give the starting address and number of identifiers occurring in this block as also the number of the immediately surrounding block. The facility in languages, of using identifiers before they are declared, makes a pre-pass essential; this pass is required to do the analysis of the block structure and declarations. It is suggested that most statements be skipped and a short grammer for declarations and block structure be used in pass 1. This however seems to have the disadvantage that identifiers (not in declarations) either have to be read again (and built up as id's) or we have to store them all in an internal area from which they will be accessed when referring the symbol table. In the scheme used by Gupta et. al. (18) . . a name table is prepared in pass-1. This table contains each identifier at only one place. As and when a declarations for an identifier is processed an attribute cell is fetched and linked in the list of attribute cells which were linked as a result of appearance of this identifier in declarations in surrounding blocks.

As discussed in section 4.4 two passes were no longer a must because of the MINIPL restriction of having variables declared before use; in fact declarations are to be the first thing in a block.

If two passes did become necessary because of core size limitation in some particular implementation, the block structured table of ALCOR (17) type could be used. The other choice (18) seems to involve more work in implementation, as a list organization is required, as well as while compiling, as pointers linking attribute cells have to be chased.

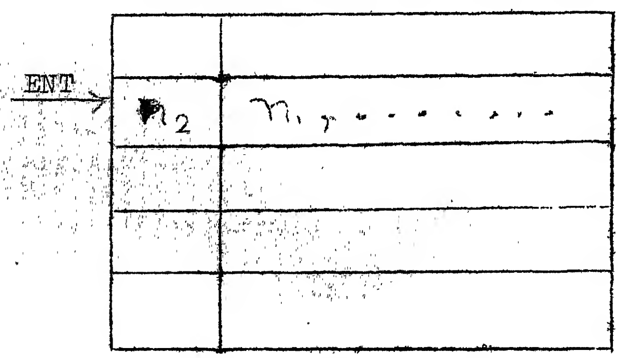
Present scheme uses a table analogous to the first scheme discussed above. We take the advantage of our one pass organization to make the searching more efficient and probably effect some economy in table size. The scheme involves, at the lexicographic opening of a block, setting a marker to the starting point, in the symbol table, at which the entires, if any, for the current block start. For every declaration a search is made upto this pointer and if the identifier is not found, an entry is made into the table. As soon as the processing of the block is over the entires for this block are deleted from the symbol table. Thus at any given time only entries corresponding to the currently active block will occupy the symbol table. The search for a use of an identifier can be made in the symbol table, last symbol backwards. The overhead in looking up the parent blocks and their addresses from the block list is now no longer there. In fact in the present one pass compiler there is no need for a block list, and consequently one is not constructed presently.

Hash addressing is relatively difficult for building symbol tables for block-structured languages. One important minus factor,

in a symbol table attribute. When block BEGIN^2 is opened and 'GO TO LL' is encountered LABCNT is increased by one,LL entered in symbol table and LABCNT value (say n2) stored as an attribute for LL. If any more 'GO TO LL' statement occur in BEGIN^2 block, the same labcnt as stored for LL(n2) is used as the operand in the quadruple generated. At the time BEGIN^2 block is closed all labels referred in it (of the LL type) are looked up in the predecessor block (BEGIN^1). If not found, they are appended at the bottom of BEGIN^1 block entires in symbol table. If found,as for the program in fig. 7.3 , then an entry is made in a separate table ; we call it LABTAB as shown in figure 7.4. The pointer to the entry ENT in LABTAB is stored as another attribute against the LL entry (for BEGIN^1) in symbol table.

The label counts for same label LL in surrounding blocks.

The lable point for LL in the beepest block.



LABTAB

Fig. 7.4

If there were blocks surrounding BEGIN^1 in which a reference to the label LL had already been made then the list of equivalent label counts would extend beyond n_1 to, say, n_0, n_{-1}...& so on.

in our context, is the size required. Seccond is the difficulty in pruning the entires from the table when blocks are closed. Considering that most references in a block will be to the entires in itself or in a few surrounding blocks, hashing probably will not be so much of an advantage, keeping in mind the overheads in hashing.

7.1.3. Description of the Symbol Table Organization in MINIPL Compiler:

We have outlined the scheme of symbol table management in MINIPL compiler already. We shall now give the details of symbol table lookup, attributes and their entering etc. Processing of declarations will also be described in short. An important thing which complicates matters is handling of labels (and equivalently, procedure names etc.). This and the modifications required by it will be taken up in the next section.

Symbol Table and its Search :

The argument column for our symbol table is an array SYMTA1 with the pointer LSYM pointing to the first empty entry in SYMTA1 (initially, 1). An identifier is stored as shown in fig. 7.1. The only difference is that first word pointed to in the string space (IDAREA) contains the number of characters (and not words - which have six characters packed in them, with trailing blonks appended to fill the last word).

The routine SRCHEN does the job of both searching the symbol table as well as entering attributes. With the argument IWHICH=1, it searches the

symbol table between the limits LOW & HIGH. The identifier is assumed to reside in a table IDTABL (necessity of which arises from the fact that more than one identifiers, which have not yet been looked up in symbol table, may at times remain in the syntax tree simultaneously), with ISTART pointing to the first entry. Search is simple and goes, after checking for equality of the length of entry in symbol table and of the identifier supplied, to compare the entries linearly from HIGH to LOW. For processing a declaration LOW is set to IBLPTR, pointer of current block into SYMTA1, and HIGH set to LSYM - 1. For looking up a non-declarative occurrance of an identifier LOW is set to 1. If the search is successful, Index is set to the position of the matching entry, if unsuccessful, to 0.

For IWHICH=2, SRCHEN enters the identifier in IDTABL at the bottom of the symbol table (LSYM). The mechanism of entering is just the reverse of accessing IDAREA for a comparision.

To store the information declared in the form of declaration **stmt** of MINIPL it is not required to reset the IBLPTR to point to the beginning in symbol table of say block A (fig. 7.2) after the block B is closed. This becomes necessary however when lakels are to be handled. For this purpose IBLPTR is saved on a stack SYMSTK. In fact it is one of a set of stacks, called collectively as secondary stacks (common area SECSTK), to store information about blocks. At the end of a block IBLPTR can be restored from the stack.

```
BEGIN     ; /*A*/
  .
  .
   BEGIN  ; /*B*/
  .
  .
   END    ; /*B*/
  .
  .
END       ; /*A*/
```

fig. 7.2

It is suggested (17) that the symbol table contain in the very beginning, all the system functions. This scheme however is not envisaged for MINIPL as function - procedures are not a part of it. System functions, which are necessary however, (like ADDR for pointer variables) may be classed as an operator SYSFN.

Attributes and their Handling :

Presently the attributes that an identifier can have been divided into eight classes (section 7.1.1). The classes and their contents are,

ATR(1) : Basic type (fixed/float/char/bit/label/procedure name)

ATR(2) : Storage class (static/external/dynamic/temporary)

ATR(3) : No of dimensions (0 for scalars)
(for labels or procedures names, whether or not the definition has been made)

ATR(4) : Procedure count

ATR(5) : Depth of procedure nesting

ATR(6) : Empty (as yet)

ATR(7) : Offset from respective data area bases

ATR(8) : Pointer to bounds area (for arrays)

The information content is not of direct relevance here ; it is to be used in the semantic processing of individual syntactic entities and is relevant there. The attributes are stored in two words (in arrays SYMTA2 and SYMTA3 respectively) for each entry in SYMTA1. First six attributes are stored in SYMTA2 and last two in SYMTA3. Routines GETATR & PUTATR fetch and store the attribute specified in NATR at the position (in SYMTA2 or SYMTA3) IPADR.

Processing of Declarations :

Here we shall just summarize the processing of declarations in short, the details can be easily gleaned from the program listings. Structures are not yet catered for, and a message saying so is printed for structural declarations. First task is to check for proper positioning of declare statements. This is done by checking, for all the declaration syntactic entities with DECLARE as the first element on the R.H.S., whether an executable statement or procedure definition has occurred so far. If yes, then call ERROR. Even though execution will be deleted, the declaration entries are made in the symbol table and program compiled to point out further errors.

A pointer IPRPTR is set in the symbol table when the first identifier of a declare list is to be entered in SYMTA1. Another pointer IENPTR keeps incrementing for each further entry entered in SYMTA1. Routine SRCHEN is called first for a look up. If

the index is nonzero, error message for multiple definition is printed. But in both the cases(for zero and nonzero index)the identifier is entered. Thus in case of multiple definition, the last definition is the one that holds for future references.

Since the factoring is single level only,all the attributes but the bound list for array identifiers, appear after the factor list is over. These attributes are than entered for all members of the factor list, in arrays SYMTA2 and SYMTA3 at addresses from IPRPTR to IENPTR. The bound list attributes are stored in temporary arrays TEMP1 & TEMP2. TEMP1 contains the number of bounds (or the dimensions), and TEMP2 the pointer to the bounds area).

In the above processing, the secondary information ITYP (FIXED/FLOAT/STATIC etc.) for the lexial unit 'TYPE' is obtained from the syntax tree TREE2. An analysis of ITYP is then made to sort out the different attributes. A point to be noted is :if a STATIC comes after EXTERNAL in the attribute list, it is not stored and the storage class remains EXTERNAL.

7.1.4. Labels, Procedure Name and External Variables:

The simple pitch of the symbol table organization described above is queered some-what when we consider the entities mentioned in the title of this section. Let us take the external variables first.

When a variable is declared with attribute EXTERNAL it has got to be entered in the current block area in the symtab. However,

it has to be linked to its occurances, if any, in declarations in other blocks, and at the end of an external procedure included in the header information. Thus the identifiers for the given block can not just be deleted from the symbol table as soon as the closing END is encountered. One method could be that a separate table is kept for external variables for the whole procedure. Now whenever, an external variable is declared in a block, it is entered in symbol table at the same time it is searched in the external-variable-table too. If found, the index is stored in a symbol table attribute. If not, it is entered in the table and index again stored in the symbol table attribute. All references to this identifier now should point to the external variable table. At the end of the external procedures this table can become a part of the header.

Label Definition.

Label references in MINIPL arise at four places (Appendix A.2):

```
<goto stmt>  →   GO TO  <label>                  (1)
<end stmt>   →  <label> .. END                   (2)
<proc heading> → <label> .. PROCEDURE
          :                                      (3)
      some variations
          .
<call stmt>  →   CALL <idproc>
          :                                      (4)
      some variations
          :
```

(2) and (3) in fact define the label as an ordinary label

(possible target of a GO TO) and a procedure name (possible target of a CALL). (1) and (4) are references to labels and procedures respectively.

Although in MINIPL we have precluded the use of variables before their declaration, the same is not true of labels. A label may be referenced before it is declared. In fact it is the only case possible in MINIPL where GO TO's are restricted to be used like effective exits, by barring any statement other than an END to be labelled. The forward references are inevitable because procedures can be recursively called. Presently we discuss the problem with reference to GO TO's and ordinary label definitions. Similar processing is required for CALL's and procedure definitions too.

It is obvious that a label definition and all references to it must be some how linked together. In FORTRAN this can be done simply by entering the first occurrance of a label in the symbol table and make it 'defined' if it occurs as a definition and as 'undefined' if it occurs in a GO TO. All the undefined quadruples are linked together and when a definition occurs all the previous ones are adjusted and the present value of label now used in future references. The linking operation does not remain as simple when block structure is taken into account.

The following example illustrates the problem :

```
        BEGIN¹;
        L : .....
            GO TO LL ;
            BEGIN² ;
              GO TO L ;
                 :
              GO TO LL ;
              L:.....
            END²;
               :
    LL: END¹;
```

Fig. 7.3

Now at the time the reference to L is made in BEGIN[2] block by a GO TO, we can not know whether to link the L to the definition in the outer block or not.

The method suggested (35) is to keep a separate table and a separate searching mechanism for labels. All the quadruples for GO TO type of references in an open block are linked together in different lists the address of which is in the symbol table entry. If a label definition occurrs in the block, all the quadruples referring to it can be corrected. If however at the end of block the label is not defined yet it has to be passed on to the surrounding block and if already there, the link of quadruples referring to the label joined with the ones in the present block.

The situation is no less complicated in MINIPL since **although the** label referred could not have been defined already in

the surrounding block (whose END will come after that of the present one). The joining of chains problem still exists for the statements of the type GO TO LL(fig. 7.3) and the undefined label entires have to be passed to the outer block. This is done at the time an END statement closing the current block is recognized. Instead of the simple popping of the symbol a routine ADJUST is called. This routine looks for all the undefined label entires, starting at IBLPTR (the pointer to SYMTA1 where the entires for the current block begin), and appends them to the previous block, if a reference does not exist there already.

An important point to mention is that giving the quadruple counter (if one is used to count the quadruples generated) setting as the value to a label, when it is defined, is considered meaningless for the following reason. Since different quadruples will translate to different number of machine code instructions, quadruple count can not be of much use to the quadruple processor; it will have to do the linking any way, unless a table is used giving the number of machine instructions to be generated for each quadruple - a rather unwieldly method.

In the proposed solution, first time a 'GO TO LL' type of reference occurrs in a block like BEGIN[1] (fig. 7.3) an entry is made in a separate table as well as the symbol table. A counter LABCNT is increased by 1 and its value (say n1) stored

At the end of an external procedures all the undefined labels will be left in the symbol table. The processing for procedure names and CALL's is similar and the undefined procedure names referred to in CALL's will be left in the symbol table. These are obviously calls to external procedures. This will also became a part of the header information.

At the time a definition for LL occurrs, quadruples defining n_2, n_1..... are issued and entry ENT deleted. Chaining of operands quadruple processor will have to be done any way as actual **machine** instruction values of quadruple will became clear only after processing is over up to the defining quadruple. However using the counts (n1 etc.) as an index in a table the need for searching in quadruple processor is eliminated.

7.2. Storage Allocation :

In MINIPL there are two classes of storage, STATIC and AUTOMATIC; the class of BASED storage is not included at present but would be required for provision of list processing facility described in the specifications of MINIPL (section 3.4.4).

7.2.1. When to Allocate Storage:

Before the final machine code is generated, run time addresses to variables must be assigned. Where to make this assignment is a major design decision. All the references in quadruples could have pointed to appropriate symbol table entries. This would delegate the task of storage allocation to the quadruple processor. **Primarily**

because this is in conflict with our aim that quadruple processor should be as simple as possible it has been decided to perform this function at compile-time itself (of course, external variables can not be allocated storage at compile time, basic entity for compilation being an external procedure. This is discussed towards the end of section 7.2). Another side advantage of performing storage allocation at compile time is the saving in the space-time product resulting from keeping the symbol table in core, only at compile time. The disadvantage is that storage allocation is machine dependent to a certain extent, but this advantage can be made less weightly by putting all storage allocation work in a separate routine (ALOCAT for the present system) and modify it, if necessary, when moving the compiler to a different machine.

7.2.2. Storage Allocation: Static, Dyanmic and Pseudo-Dynamic

Static storage can be allocated to the variable so declared, in an area for the whole external procedure; of course, scope of the particular variables is determined by the block structure. The AUTOMATIC storage class is applicable to the non-static variables in various procedure and begin blocks. Although storage of these can be allocated statically (PL/I definition says that variables of a block or procedure may not have the same values between invocations, but then it is not required to destroy the values), we can effect a saving in storage by allocating storage dynamically. The dynamic allocation is, anyhow, a must for recursive procedures.

In the present implementation compiler allocates storage dynamically for procedures. This includes obtaining the offset of variables from the base of the data area, which is allocated at run time on a stack and the size of which calculated at compile time. For begin-blocks a pseudo-dynamic scheme is adopted as the addressing (at run time) in the dynamic data area is considerably more cumbersome than addressing in the static area. The scheme involves considering the run time data-area as a stack for block-areas However the allocation for blocks is not made at run time. Instead, at the compile time itself a pointer pointing to the top of the storage allocated so far (within the procedure data area) is decremented when a block is closed by the amount of storage allocated to this just-closed block. Scheme will be clear from the example given below :

```
A1:PROCEDURE ;
   .
   .
  A2:PROCEDURE ;
     .
     BEGIN1;
       .
       BEGIN2;...END;
       BEGIN3;...END;
     END;
     BEGIN4; ... END;
  END ;
END;
```

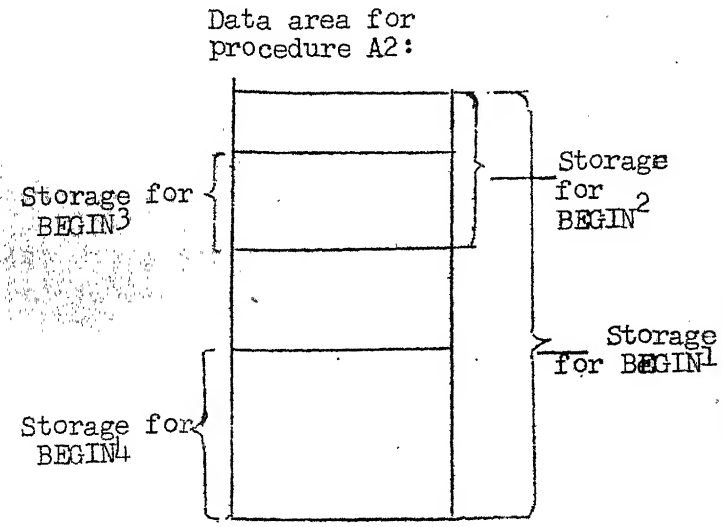


Fig. 7.5

Thus parallel blocks can be allocated storage one over the other.

7.2.3. The Present Algorithm:

The storage allocation is done by the routine ALOCAT, which is called when ever storage is to be allocated to a variable. Before calling, the attributes of the variables are accessed from the symbol table and put in the array ATR. Routine ALOCAT basically maintains two pointers STOR(1) and STOR(2); for dynamic and static storage allocated so far. Whenever a call is made to allocate a variable storage in static area, STOR(2) is incremented by the amount of locations needed for the variable in question (a refinement will be given in next sub-section). For dynamic allocation same is done to STOR(1). The difference however is in adjustment of STOR(1) and STOR(2) in the routine SEMAN. STOR(2) is untouched in the whole external procedure and gives the total static area needed for the external procedure. STOR(1) on the other hand is treated as follows:

a. For every block begining (BEGIN;) the value of the STOR(1) is saved on a secondary stack STORSK (done in routine SAVRES).

b. For every procedure heading the value is saved as in (a). The STOR(1) now is set to 0.

c. When the block is ended, the secondary stack is popped and the STOR(1) restored.

d. Restoring is done as in c, but prior to that the STOR(1) value (total size of data-area required by the procedure)

is set as the initial value of a system generated variable (G. proc count), which is used for run time allocation (section 7.3). In proc count is the lexicographic count of the procedure, i.e. the order in which it was opened, and is associated with the procedure entry through a third secondary stack (LABBLK).

What all gets Allocated in a Data Area:

To implement the addressing mechanism (section 7.3) and the procedure linkage i.e. information and control flow between procedures certain addresses are to be allocated in the data area. All this is taken care of by calling ALOCAT at appropriate places (i.e., while processing the procedure headings and CALL's etc.). Of course, alocate is called when processing declarations to allocate storage for different variables.

Refinements in ALOCAT :

Though the basic purpose of the storage allocation is served by the description given above. Some refinements were made in ALOCAT to take care of another problem. In many machines, alighment of variables on certain boundaries (e.g. byte, half ward, full word etc.) is required. With the previous mechanism, the storage allocator ALOCAT would pad the previous value to bring the new address at the proper boundary. This will cause a lot of wastage as the variables could come in an arbitrary order. Gries (17) has suggested that addresses should be assigned first to all words which require alignment on say a

double word boundary then to the ones requires say a word boundary, half word boundary & so on. The scheme is possible in MINIPL where all declarations come before executable statements. However, the implementation becomes clumsy as many counters are to be maintained (and their values stored in symbol table temporarily), then at the end of declarations all equivalent addresses in data area calculated (and the counts in symbol table attributes replaced appropriately). The method, that has been incorporated in ALOCAT, to attack the problem is as follows (given only for STOR representing both STOR(1) and STOR(2) processing for which is identical).

STOR is set initially to 1-MAXUNT where MAXUNT is the maximum no of the smallest addressable units (e.g. byte for the IBM 360) required for the largest TYPE (floating pt. in our case). For each type there is a counter which indicates the position within the MAXUNT cell for that type. When a call for allocation is made, the position pointer is incremented by the no of units required for the particular TYPE, and the address calculated. If the MAXUNT cell is full, a new MAXUNT cell is allocated by increasing STOR and position counter set to the begining of it.

The method given above assumes that all boundaries fall on the MAXUNT boundary. If it is not so, the least common multiple of all the word boundaries will have to replace MAXUNT above. Although the above method is no advantage to the IBM 7044 implementation (where the smallest, and also the largest, unit is word) it was

written to cater for the byte organized machines. The parameterization of the routine was attempted to make it machine independent but soon it was realized that to pack data into the smallest addressable unit, will require a rather complex routine and such modifications are best (most efficiently) incorporated for the individual hest computer.

Initialisation :

So far the allocation mechanism just allocated the storage to variables for run time. The information available was the length of data-areas and it was sufficient. However, to allow initialisation of variables, additional work is needed. One method is to generate assignment statements for the initial statements which are executed at the begining of run time. Another method and the one envisaged here is to prepare a table of initial values and the corresponding addresses and use the quadruple processor to generate data definition machine instructions. In addition to the initialised variable, the above scheme is necessary at times to put information needed at the time of generating a quadruple (say the size of data area, needed for the GETREA quadruple (sec. 7.3)) and becomes available only later (the size of the data area becomes known only at the end of a procedure).

7.2.4 Storage Allocation and the Quadruple Processor:

The information, in addition to the quadruples generated, supplied to the quadruple processor will be,

a. The table of external references procedures and variables.

b. The table of the initialization values (this will include the sizes of the procedure data areas etc. too).

The operands of the quadruple processor contain for dynamic area variables, the procedure level no. (section 7.3) and the offset within the procedure. For these the quadruple processor has to generate machine instructions to access them (section 7.3). For other types of storage the first subfield of an operand field contains the indication of the type of information to follow in the second subfield (details of the quadruple format are given in Chapter 7 (part 2)). The second field will contain the offset in the static area, position in the external table and pointer to the label table etc.

Use of IBMAP (the MAP assembler on IBM 7044) was envisayed to process quadruples for the present implementation (section 4.6.3). In addition to the awkwardness pointed out in section 4.6.3, the MAP assembler (representative of most conventional assemblers for processing assembly languages which are intended for programming work too) is awkward to take advantages of the processing done in compiler phase. As will be seen in section 7.3 an offset n1 in static area is to be translated as S.n1, thus repeating the symbol table formation and storage allocation. Similarly, while a special purpose quadruple processor would build the initialization values in the static area, at present we will have to issue define instructions

at the end. Presently, an instruction to reserve storage for the static area for the particular program is issued , while the special quadruple processor could place this information in the header of the external procedure. This would have made possible, for a linking loader, if it had the capability to do so, to put the static storage of all the procedures together, thereby separating the program area from the data area totally. As for the construction of the header it was already over as the operands referenceing external variables and procedures could have pointed to the header table (a). To use map,references by name have to be generated and at the end of compilation, for all the entries of table (a), EXTERN instructions generated.

7.3. Addressing Mechanism and Run Time Storage Administration.

The detailed format of quadruples has been discussed in Chapter 7 (II) and the addressing for static and external variables already explained. Examples of both can be seen from the Addfix macro (Appendix E) where the operand subfield 1 indicates external variable (100) and static variable (101). Presently we shall discuss the addressing for dynamic area.

7.3.1. Addressing in Dynamic Area:

In MINIPL any procedure can contain a nonlocal reference to a variable declared in a surrounding procedure (and not declared in the procedure in question. Though the (lexicographic-count, offset) pair defines a variables completely. The present

mechanism makes use of the depth of nesting (the level) of the procedure as the first component of the subfield pair for a dynamic area operand. This hierarchy number (level) can be used as the procedures which have the same hierarchy number are in parallel blocks, therefore the compiler never processes them at the same time. Primarily the effective address of a dynamic variable (k,i) can be obtained by adding to base address of the data-area for procedure at level k , the offset i.

7.3.2. Run Time Storage Administration :

The first executable quadruple to be produced for a procedure heading is GETREA, $\underbrace{\text{I}}_{\text{op 1}}$, $\underbrace{\text{G.Proc count}}_{\text{op 2}}$ where I is the level of the procedure and G. Proc count described in initialization sub-section of section 7.3. This quadruple is written as a macro, which is executed when a procedure is invoked. The macro alocates storage equal to G. Proc count in a stack (Appendix F) STACK. A gloabal location STKTOP contains the top of the stack above which a new allocation is made. The base address of the procedure data area is placed in index register 1. Now a reference to a variable in the currently active procedure could be done using the indexing facility of IBM 7044, e.g.,

```
OPCODE        OFFSET,1
```

For this purpose the reference within the procedure are indicated by a special first component (=102) of the operand subfield-pair.

To make references out side the procedure (to surrounding procedures), the base addresses of the referrable procedures are copied from the calling procedure data area, in the first few locations of the data-area allocated.

Two cases arise (fig. 7.6);

```
Proc A
  Proc B
    CALL  C    - (1)
  Proc C
    Proc D
      CALL B   - (2)
```

Fig. 7.6

In the first case (e.g., B invoking C-(1)) both the procedures are at the same level i. Then both can refer to the procedures at level 0,1,.....,i-1. Thus GETREA just need copy the first i locations into the new data-area.

In the second case (e.g., D invoking B - (2)), the called procedure has level lower than the calling procedure but is declared in the surrounding block. Again both the procedures can refer to procedures at level 0,1,.....,i-1. In addition, the calling procedure can refer to some more procedures,but that is immaterial. Again, we copy the first i locations from the data area

of the calling procedure into the new data area. The base address of the new data area is plugged in (i+1)th location.

The whole process starts by executing in the main procedure the macro SETREA, instead of GETREA (Appendix F). SETREA allocates storage to main procedure and copies its own address in the first location ((i+1)th locn, where i = 0).

Before exiting from the procedure, macro FRIREA is executed. This returns the data area to the run time stack and adjusts STKTOP etc. The **macros** may include instruction for saving return address and executing the return from procedure (as for the macros given in Appendix F). These may also include the handling of information flow through formal parameters.

7.3.3. Addressing and Instruction Set of IBM 7044 :

Let us look at the sample instructions generated (Appendix E) for the operand part for a dynamic area reference (k,i) where k indicates the level of a surrounding procedure.

The instructions are of the form,

(1)	STO	ADTEMP	(Save accumulater into a temporary location)
(2)	CLA	k,1	(load the base address of the procedure at level k into the accumulater)
(3)	PAC	,2	(Put the best address in index register 2)

(4)	OP	i,2	(access dynamic area)
(5)	CLA	ADTEMP	(restere the accumulator from the temporary location)

The above shows the weakness of an instruction set which was not designed for dynamic addressing. The use of accumulator and associated over head (instr. (1) and instr. (5)), to load the base address into register two comes because there is no concept of base registers in IBM 7044. We are using index registers as base registers (which is a crime, any way, as it comes in the way of efficient use of index registers for indexing (unless the save-restere is kept track of which is an over head by itself). There is an instruction (LAC) for loading an index register from storage but in this the source address can not be indexed itself.

If we do not wish to use index registers the following sets of instructions could be generated:

a) Only index register (1) used for always holding the current data area base address.

(1)	STO	ADTEMP
(2)	CLA	k,1
(3)	ADD	= i
(4)	STA	INDADR
(5)	OP*	INDADR
(6)	CLA	ADTEMP

Now the temporary location INDADR is used for addressing (the instruction 3,4,5 have changed). The number of instructions remains the same , but, one memory reference has increased (instr. (5)).

b) Index register 1 also not used. A global location ACTREA holds the current base addr.

```
(1)    STO    ADTEMP
(2)    CLA    ACTREA
(3)    ADD    = i
(4)    STA    INDADR
(5)    OP*    INDADR
(6)    CLA    ADTEMP .
```

The last scheme seems best as it avoids using index registers completely and is not significantly worse than the first. However, the reference to the same area will also now take (6) instructions instead of one,

```
OP    k,1
```

Thus among the three the first scheme, also used in the macro in Appendix F, seems the most satisfactory.

7.4 Conclusions:

In this section we described various aspects of the three areas mentioned in the title of the section. Not everything has been implemented. In particular, the quadruple generation for different semantic routines is not implemented, (i.e. label definitions generator of area management macros etc. The basic organization, however, has been implemented.

CHAPTER 8

DISCUSSION

Looking back at our work, with the benefit of hindsight we now have, some observations about the project can be made.

Considering the language design part, we observe that the approach has been mostly empirical. It is based upon the important ideas and observations about the psychology of programming. Although some of the ideas about program structure are less vague, yet there is a considerable amount of difference of opinion. Moreover, it is difficult to say conclusively, how good a match MINIPL is with the needs outlined earlier. The problem of simulating the desired constructs and the nonavailability of case are irritants which have been tolerated to quickly get to a workable language. By adopting the format of PL/I, compatibility was achieved with a fairly wide spread and well supported existing language, which is a definite advantage. Only usage of the language, if it is fully implemented, can say how well it meets the needs of the users. However, the identification of the criteria for picking the subset is important by itself.

The material in the implementation portion, as a look will show, has wide variation in its form and presentation.

The primary reason is the natural variety of information and the different levels at which different things can be presented. A description of individual routines is desirable and feasible when the said routine performs a single task. This however, is not always true. Certain basic tasks can be identified but their handling is distributed over the whole program. Semantic routines are an example. While processing for an END statement, a host of actions related to, say, the symbol table management, storage allocation etc., may be taken. The approach has been to give more information about a particular basic function. With the huge amount of information that can be needed it may be true that documentation is less than complete. However, it has been attempted to give the logic by discussing the problems and the strategies with description of routines just serving to fix ideas.

Another thing that must be pointed out is that, as far as implementation description goes, we have talked about things which are at various stages of completion. The lexical and syntax analysers have been fully implemented. In addition, the overall structure for semantic processing has also been developed and implemented. With this, after getting a general feel of the syntax analysis and general semantic analysis, it should not be too difficult to extend the semantics to include other facilities too. The basic semantic routines

like symbol table management, storage allocation, input-output and the generation of quadruples, have also been implemented. Description of certain things, like label handling, quadruple processing etc., details of which were chalked out, have been included although their implementation has not been carried out. Certain features of MINIPL have not been included, in the running operator grammer, though efforts were made which pointed the way to the discussion of problems in using transition matrix technique in Chapter 5. The resulting transition matrix, however, was too large to be used without packing more than one element per word. This is not presently incorporated.

At times problems arose because the constraint of time prevented the desirable separation of design and coding of the whole program even though it is agreed that such an approach is beneficial in the long run when total implementation is the goal.

The aim of the project was to experiment with the translation process using syntax techniques and with additional constraints of machine independence etc. and it can be said that it has been achieved to some extent. Using transition matrix gave insight into the difficulties of making the grammar of a real life language suit a particular algorithm. Implementation of semantics was indicative of the complexities involved.

Finally, it is hoped that the information about our experience in this project will be useful to others undertaking similar projects.

REFERENCES

1. Ashcraft E. and Manna Z., 'The Translation of Go-To Programs to While Programs', Stanford Research Report, SR-70.

2. Beizer B., "The Architecture and Engineering of Digital Computer Complexes", Plenum Press, 1971.

3. Bochmann G.V., 'Multiple Exits from a Loop without GO TO', CACM 16 (July 1973).

4. Dennis J.B., 'Modularity', 'Advanced Course in Software Engineering', Ed. Bauer F.L., Spring Verlay, Lecture Notes in Economics and Mathematical Systems - 81 (1973), pp 128-182.

5. Dijkstra E.W., 'The Humble Programmer', CACM 15 (November 1972) pp 859-866.

6. Dijkstra E.W., 'A Short Introduction to the Art of Programming', Lecture Notes, Technishe Hogeschool Eindhoven.

7. Dijkstra E.W.,Hoare C.A.R., Dahl O.J., 'Structured Programming', Academic Press, 1972.

8. Dijkstra E.W., 'GO TO Statement Considered Harmful', CACM 11 (March 1968), pp 147-148.

9. Early J., 'An efficient context free parsing algorithm' CACM 13 (Feb. 1970) pp 94-102.

10. Evans, A., 'An Algol-60 Compiler', Annual Review in Automatic Programming, 4 (1964), pp 87.

11. Evans J.D., Jevans D., ed.,'Software 70', Auerbach Publishers, 1970.

12. Feldmann J.A., Gries D., 'Translator Writing Systems', CACM 11 (Feb. 1968), pp 77.

13 Floyd R.W., 'The Syntax of Programming Languages - a Survey'. IEEE Trans. EC 13, 4 (Aug. 1964), 346-353.

14. Goos G., 'Programming Languages as a Tool in Writing System Software', 'Advanced Course in Software Engineering', Ed. Bauer F.L., Spring Valley, Lecture Notes in Economics and Mathematical Systems - 81 (1973), pp 47-69.

15. Gries D., 'Use of Transition Matrices in Compiling', CACM 11 (Jan. 1968), pp. 26-34.

16. Gries D., 'Compiler Construction for Digital Computers', Wiley International Edition, 1971.

17. Gries D., Paul M., and Weihle H.R., 'Some Techniques used in the AICOR ILLINOIS-7090' CACM 8 (Aug. 1965), pp 496-500.

18. Gupta A., Kannan K., Saha A.B., Sahasvabuddhe H.V.,'PL/I on IBM 7044', Computer Center, I.I.T., Kanpur.

19. IBM System/360 Operating System, PL/I(F) Language Reference Mannual, Order No. GC 28-8201-4.

20. Clint M. and Hoare, C.A.R., Program Proving: Jumps and Functions, Acta Jupermatica 1(1972) p. 214.

21. Knuth D.E., 'A review of Structured Programming', Technical Report, Stan - CS - 73 - 371.

22. Naur P. and Backus J.W., et. al., 'Revised Report on the Algorithmic Language Algol 60', CACM 6, (Jan. 1963), pp 1-17.

23. Patil R.S., 'Syntax Analysis in MTS', M. Tech. Thesis, Depart. of Elect. Engg., IIT Kanpur (April, 1972).

24. Perlis A., 'A Survey of Programming and Programming Languages', The Jerusalem Conference on Information Technology Proceedings, Aug. 71 pp. 192-207.

25. Peterson W.W., Kasami T., and Tokura N.T., 'On the Capabilities of <u>While</u> <u>Repeat</u> and <u>Exit</u> Statements', CACM 16 (Aug. 1973) pp 503-512.

26. Pollack S.V. Sterlig T.D., 'A guide to PL/1', Honlt, Reinhart and Winston, 1969.

27. Ramarao T.P., 'Lexical Processor and Servicing Routines for MTS', M. Tech. Thesis, Depart. of Elect. Engg., IIT Kanpur (March, 1972).

28. Rosin R.F., 'Teaching about Programming', CACM 16 (July 1973), pp. 435-438.

29. Rosin R.F., 'Programming Languages: Technical Overview', The Jerusalem Conference on Information Technology, Proceedings, Aug. 71 pp. 216-229.

30. Samelson K. and Bauer, F.L., 'Sequential Formula Translation', CACM 3 (Feb. 1960), pp 76-83.

31. Steele C.A., Sedgewick A.E., 'DEFT a desciplined Extension of Fortran', DCS, Univ. of Toronto, Technical Report No. 62, Feb. 1974.

32. Stone H.S., 'Introduction to Computer Organization and Data Structures', McGraw Hill, 1972.

33. Tsiehritzis D., 'Reliability', 'Advanced Course in Software Engineering', Ed. Bauer F.L., Spring Verlag, Lecture Notes in Economics and Mathematical Systems-81 (1973) pp 319-371.

34 Weinberg G.M., 'The Psychology of Computer Programming', Van Nostrand Reinhold Co., 1971.

35. Wirth N., 'PL360, A Programming Language for the 360 Computers', JACM 15, (Jan. 1968) pp 37-74.

36. Wirth N., 'Systematic Programming', Prentice Hall, Inc., 1973.

37. Wulf W.A. 'Programming without the GO TO', IFIP Proceedings, (1971), pp 408-413..

38. Wulf, W.A., Russel D.B. and Habermann A.N., 'BLISS: A Language for Systems Programming', CACM 14, (Dec. 71) pp 781-790.

A.1 : SYNTAX OF MINIPL

The syntax for the lexical units for MINIPL in an easily readable form, is given and followed by the syntax of the MINIPL language, the lexical units forming its terminal.

The syntax notations used are as follows:

1. Braces {} are used to denote grouping. A vertical stroke is used to indicate that a choice is to be made.

   e.g. { FIXED FLOAT }

   indicates the occurance of either FIXED or FLOAT.

   { }* is used to indicate that the group may occur once, more than once, or not at all { }*1 indicates at least one occurance of the group.

2. Square bracket [ ] denote options. Anything enclosed in brackets may appear once or may not appear at all.

3. The symbols for lexical units appear in bold type.

Basic elements of lexical units :

letter	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
digit	0 1 2 3 4 5 6 7 8 9
quote	'
others	+ - * / = $, . ( ) ␢

The lexical units :

identifier (ID) — letter {letter |digit} *

integer (INTEGER) — {digit}^{*1}

floating point no. (FLTPT) — {{digit}^{*1} . {digit}* |{digit}* .
{digit}^{*1} } E [+|-] digit [digit]

boolean constant (BOOLCON) — {0|1} B

Character constant (CHARCON) — ' {letter | digit |{quote quote}| others}'

char string (STRING) — '{letter| digit|{quote quote}|others}
{letter|digit|{quote quote}|others}^{*1} '

delimiters — {+|-|*|/|=|$|,|.|(|)|**|,.|.. }

The MINIPL Language :

variable — {ID | ID (arith exp {, arith exp }*) |
{ID| ID(arith exp{ ,arith exp} *)} .
{ID| ID(arith exp{ , arith exp}*)}

afactor — { aprim| afactor ** aprim}

aprim — { INTEGER | FLTPT|variable| (arith exp) }

arithexp — { [+|-] aterm | arith exp +|- aterm}

bool prim — { BOOLCON | variable| (bool exp)}

bool fac — {bool sec |bool fac AND bool sec }

bool exp — {bool fac| bool exp OR bool fac}

bool sec — {bool prim |NOT bool prim|rel exp }

char prim	{ CHARCON|variable}
relational exp	{arith exp {= |NE|GT|NG|GE|LT|NL|LE}arith exp| {char prim {= |NE }char prim|bool prim {=|NE|} char prim}
expression	{arith exp|bool exp|char prim }
return stmt	RETURN,.
begin	BEGIN ,.
stop stmt	STOP ,.
assign stmt	var = exp,.
end	{END| ID.. END},.
call stmt	CALL ID[(exp {,exp}*)],.
proc head	ID.. PROCEDURE[(ID {, ID.})][RECUR],.
get list stmt	GET LIST (variable {, variable }*),.
get edit stmt	GET EDIT (variable {, variable }*) format,.
put list stmt	PUT [PAGE|SKIP [INTEGER]]LIST ({exp| STRING } {,{ exp| STRING }}*),.
put edit stmt	PUT[PAGE|SKIP [INTEGER]]EDIT({exp|STRING}{ , {exp |STRING }}*) format,.
format type	{{A|X|F|B|} (INTEGER)| E(INTEGER,INTEGER)| SKIP (INTEGER)}
field	[INTEGER]format type
group	INTEGER (field {, field }*)
format	({group| field}{ ,{ group| field }}*)
attribute	{FIXED|FLOAT|BIT|CHAR|EXTERNAL|STATIC|BASED }

```
attrlist      attribute { , attribute}*

elem          ID[ (INTEGER.. INTEGER{ , INTEGER..INTEGER}*)]

elemlist      elm { , elem}*

decl stmt     DECLARE {1 elem {, 2 elem attrlist }*1|

              {elem |elemlist} attrlist

              { ,{ elem|elemlist} attrlist }* } ,.

null           ,.

if stmt        IF boolexp THEN stmt[ ELSE stmt]

stmt          { null|getlist|getedit|putlist|putedit|stop |

               return|assign|call|ifstmt|group}

dohead         { DO|DO var ={ integer|variable} To{integer|

                variable} [BY  {integer|variable }]|DO WHILE

                boolexp },.

group          {dohead|begin}{stmt}* end
```

A.2 Operator Grammar for MINIPL:

1.	<subscr head>	→	<id> (<arith exp>
2.	<subscr head>	→	<subscr head> , <arith exp>
3.	<subs var>	→	<subscr head>)
4.	<struc element>	→	<subs var>
5.	<struc element>	→	<id>
6.	<struc var>	→	<struc element> . <struc element>
7.	<id amb>	→	ID
8.	<id>	→	<id amb>
9.	<id dec>	→	<id amb>
10.	<id formal>	→	<id amb>
11.	<id proc>	→	<id amb>
12.	<label>	→	<id amb>
13.	<var>	→	<subs var>
14.	<var>	→	<id>
15.	<var>	→	<struc var>
16.	<arith exp>	→	<arith term>
17.	<arith exp>	→	ADOP <arith term>
18.	<arith exp>	→	<arith exp> ADOP <arith term>

19.	<arith term>	→	<arith factor>
20.	<arith term>	→	<arith term> MULOP <arith factor>
21.	<arith factor>	→	<arith prim>
22.	<arith factor>	→	<arith factor> ** <arith prim>
23.	<arith prim>	→	<int const var>
24.	<arith prim>	→	FLTPT
25.	<arith prim>	→	(<arith exp>)
26.	<arith prim>	→	<var>
27.	<int con var>	→	INTEGER
28.	<int con var>	→	<var>
29.	<bool prim>	→	BOOLCON
30.	<bool prim>	→	<variable>
31.	<bool prim>	→	(<bool exp>)
32.	<relation exp>	→	<arith exp> RELOP <arith exp>
33.	<relation exp>	→	<char prim> RELOP <char prim>
34.	<relation exp>	→	<bool prim> RELOP <bool prim>
35.	<char prim>	→	CHARCON
36.	<char prim>	→	<var>
37.	<bool secndry>	→	<bool prim>
38.	<bool secndry>	→	NOT <bool prim>
39.	<bool secndry>	→	<relation exp>
40.	<bool factor>	→	<bool secndry>
41.	<bool factor>	→	<bool factor> AND <bool secndry>
42.	<bool exp>	→	<bool factor>
43.	<bool exp>	→	<bool exp> OR <bool factor>
44.	<exp>	→	<arith exp>

45.	<exp>	→	<bool exp>
46.	<exp>	→	<char prim>
47.	<call list>	→	CALL <id proc> (<exp>
48.	<call list>	→	<call list> , <exp>
49.	<proc list>	→	<label> •• PROC (<id formal>
50.	<proc list>	→	<proc list> , <id formal>
51.	<get list stmt>	→	GET LIST (<get list>)
52.	<get list>	→	<var>
53.	<get list>	→	<get list> , <var>
54.	<get edit>	→	GET EDIT (<get list>)
55.	<put list stmt>	→	PUT LIST (<put list>)
56.	<put list stmt>	→	PUT <format 1> LIST (<put list>)
57.	<put edit>	→	PUT <format 1> EDIT (<put list>)
58.	<put edit>	→	PUT EDIT (<put list>)
59.	<get edit stmt>	→	get edit (<format>)
60.	<put edit stmt>	→	<put edit> (<format>)
61.	<put list>	→	<exp>
62.	<put list>	→	STRCON
63.	<put list>	→	<put list> , <exp>
64.	<put list>	→	<put list> , STRCON
65.	<format 1>	→	PAGE
66.	<format 1>	→	<skip>
67.	<skip>	→	SKIP (INTEGER)
68.	<skip>	→	SKIP
69.	<format>	→	<group fld>

70.	<format>	→	<format> , <group fld>
71.	<group field>	→	<field type>
72.	<group field>	→	INTEGER <field type>
73.	<group field>	→	<integer> (<format>)
74.	<field type>	→	F (INTEGER)
75.	<field type>	→	F (INTEGER , INTEGER)
76.	<field type>	→	<skip>
77.	<bound list>	→	<bound list> ,<exp> •• <exp>
78.	<bound list>	→	<exp> •• <exp>
79.	<decl stmt>	→	<decl nonstrc stmt>
80.	<decl stmt>	→	<decl strc stmat>
81.	<decl prfx>	→	DECLARE (<id dec>
82.	<decl prfx 1>	→	DECLARE (<id dec> (<bound list>
83.	<decl prfx 1>	→	<decl prfx 1> , <id dec>
84.	<decl prfx 1>	→	<decl prfx 1>,<id dec> (<bound list>)
85.	<decl prfx 1>	→	<decl nonstrc stmt> , (<id dec>
86.	<decl prfx 1>	→	<decl nonstrc stmt> , (<iddec> (<bound list>)
87.	<decl nonstrc stmt>	→	<decl prfx 1>) TYPE
88.	<decl nonstrc stmt>	→	<decl nonstrc stmt> TYPE
89.	<decl nonstrc stmt>	→	<decl prfx 2> TYPE
90.	<decl prfx 2>	→	DECLARE <id dec>
91.	<decl prfx 2>	→	DECLARE <id dec> (<bound list>)
92.	<decl prfx 2>	→	<decl nonstrc stmt> , <id dec> (<bound list>)
93.	<decl prfx 2>	→	<decl nonstrc stmt> , <id dec>

94. <decl strc prfx 1> → DECLARE INTEGER <id dec>

95. <decl strc prfx 1> → DECLARE INTEGER <id dec> (<bound list>)

96. <decl strc prfx 2> → <decl strc prfx 1> , INTEGER <id dec>

97. <decl strc prfx 2> → <decl strc prfx 1> , INTEGER <id dec> (<bound list>)

98. <decl strc prfx 2> → <decl strc stmt>

99. <decl strc stmt> → <decl prfx 2> TYPE

100. <proc heading> → <label> .. PROCEDURE RECUR

101. <call stmt> → <call list>)

102. <call stmt> → CALL <id proc>

103. <main proc heading> → <label> .. PROCEDURE OPTIONS (MAIN)

104. <proc heading> → <label> .. PROCEDURE

105. <proc heading> → <proc list>)

106. <stop stmt> → STOP

107. <end stmt> → END

108. <end stmt> → <label> .. END

109. <go to stmt> → GO TO <label>

110. <begin stmt> → BEGIN

111. <return stmt> → RETURN

112. <cond stmt> → IF <bool exp>

113. <if then stmt> → Φ <cond stmt> THEN

114. <else stmt> → Φ ELSE

115 <null stmt> → Φ ,.

116. <do> → DO

117. <do stmt> → DO

118. <do stmt> → DO <id> RELOP <int con var> TO <int con var>

119.	<do stmt>	→	DO <id> RELOP <int con var> TO BY <int con var>
120.	<while stmt>	→	<do> WHILE <bool exp>
121.	<assign stmt>	→	<var> RELOP <exp>
122.	<stmt>	→	Φ <get list stmt> , .
123.	<stmt>	→	Φ <get edit stmt>
124.	<stmt>	→	Φ <put list stmt>
125.	<stmt>	→	Φ <put edit stmt>
126.	<stmt>	→	Φ <decl stmt>
127.	<stmt>	→	Φ <call stmt>
128.	<stmt>	→	Φ <main proc heading>
129.	<stmt>	→	Φ <proc heading>
130.	<stmt>	→	Φ <stop stmt>
131.	<stmt>	→	Φ <end stmt>
132.	<stmt>	→	Φ <go to stmt>
133.	<stmt>	→	Φ <begin stmt>
134.	<stmt>	→	Φ <return stmt>
135.	<stmt>	→	Φ <do stmt>
136.	<stmt>	→	Φ <while stmt>
137.	<stmt>	→	Φ <assign stmt>
138.	<proc heading>	→	<proc list>) RECUR
139.	<integer>	→	INTEGER
140.	<group field>	→	(<format>)

A.3 : CODING SCHEME FOR THE CONSTRUCTOR

The input grammer to the constructor must be an operator grammar expressed in the BNF form and coded in numbers for non-terminals and terminals to form a productions matrix. The minimum number of columns in the matrix must be six. Coding for the non-terminals should start from number 1 onwards. Coding for terminals can be started from any number beyond the greatest code for non-terminals. The number, KTT from which onwards coding for terminals starts should be specified alongwith the number of non-terminals, KNT, the number of terminals KT, the maximum number N, of terminals and non-terminals in a production and the total number M, of productions in the grammar.

The operator grammar for MINIPL is given in A.2. The values of KTT, KNT, KT, N, M and the codes used for the non-terminals and terminals are given below.

KTT	100
KNT	63
KT	46
N	9
M	140

The codes used for the non-terminals of the grammar for MINIPL are given below:

Code	Non-terminal	Code	Non-terminal
1	<subscr head>	22	<put list>
2	<subs var>	23	<format1>
3	<strc element>	24	<format>
4	<id>	25	<group fld>
5	<var>	26	<field type>
6	<arith exp>	27	<decl prfx1>
7	<arith term>	28	<bound list>
8	<arith factor>	29	<decl nonstrc stmt>
9	<arith prim>	30	<decl prfx2>
10	<int con var>	31	<decl strc prfx1>
11	<bool prim>	32	<decl strc prfx2>
12	<relation exp>	33	<decl strc stmt>
13	<char prim>	34	<cond stmt>
14	<bool secndry>	35	<do>
15	<bool factor>	36	<get list stmt>
16	<bool exp>	37	<get edit stmt>
17	<exp>	38	<put list stmt>
18	<call list>	39	<put edit stmt>
19	<proc list>	40	<decl stmt>
20	<label>	41	<call stmt>
21	<get list>	42	<main proc heading>

Code	Non-terminal
43	<proc heading>
44	<stop stmt >
45	<end stmt >
46	<go to stmt >
47	<begin stmt>
48	<return stmt >
49	<do stmt >
50	<while stmt >
51	<assign stmt >
52	< stmt >
53	< else stmt >
54	< null stmt >
55	< get edit >
56	< put edit>
57	< struc var>
58	< skip>
59	< iddec >
60	< id formal >
61	< id proc >
62	< idamb>
63	<integer >

The codes used the terminals of the grammar for MINIPL are given below:

Code	Terminal
101	.
102	MULOP
103	,
104	NOT
105	RELOP
106	AND
107	OR
108	(
109	)
110	ADOP
111	**
112	;.
113	..
114	F
115	END
116	DO
117	IF
118	THEN
119	ELSE
120	WHILE
121	GO
122	TO

Code	Terminal
123	BY
124	DECLARE
125	TYPE
126	STOP
127	BEGIN
128	GET
129	PUT
130	LIST
131	EDIT
132	SKIP
133	PAGE
134	CALL
135	RETURN
136	PROCEDURE
137	OPTIONS
138	MAIN
139	STRINGCON
140	INTEGER
141	FLTPT
142	CHARCON
143	BOOLCON
144	ID
145	ϕ
146	RECUR

APPENDIX B

IMPORTANT COMPILER DATA AREAS AND TABLES

EDTFLG	This flag is set while processing a format.
STAKI	Contains U*s during syntax analysis.
STAK2	Contains the range of the future non terminal.
NL	Is the non-terminal on top of the stak.
TREE1	Syntax tree. It contains the relevant terminals and nonterminalswhich form the syntax tree.
TREE2	Syntax tree. Corresponding to each entry in Tree1, there is an entry in TREE2 which points to a table which gives further information about the entry (e.g. it will point to an entry in the entry in the symbol table if the TREE1 entry is a variable).
TREE3	It contains the data type of the corresponding element in TREE1, or a zero if the element is a variable.
IDTABL	Identifiers from the lexical unit are put in it. Structure is similar to that of STRING.
USTAR	An array, it's nth entry gives the upper range. in TERM1 for search by GETSBN corresponding to nth U*.
TERM1	Contains terminals which may form pairs with the U* in whose range they lie.

TERM2	It gives the upper range in U1 in which GETSBN makes a search to find a U corresponding to the pair U* T
U1	It contains U's such that a reduction exists for the U* T pair and any of these U's.
U2	It contains the rule number of the reduction to be performed.
U3	It contains type of reduction to be performed.
ITRA	It is the transition matrix which represents the fsa.
ITEM2	It contains the characters of the lexical unit, one character per word in the lowest order bits.
OUTPUT	It is the output matrix corresponding to the states of the transition matrix.
IC	It is the code of the terminal returned by the lexical.
IC2	It is the subcode for the terminal. For example if IC indicates that the non-terminal is ADOP then IC2 tell whether it is '+' or '-'.
LITEM	It is the length of the lexical unit .
MEMB.	This array is accessed using the internal code of the character to get its class.
STR	This is an array into which the routine PACK packs the characters of ITEM2.
RSRTBL	It is an array of reserved words. Some of the reserved words occupy two words of the array.

STRING	This array contains the character string constants. Each string constant is preceded by a word which contains the number of charactersof the string to a word.
INTABL	Table in which fixed point constants are entered.
INTBL2	Allocated address of the fixed point constants in INTABL are put in this table.
FLTABL	Table into which floating point constants are entered.
FLTBL 2	Allocated address of the floating point constants in FLTABL are put in this table
BITBL2(2)	First ontry contains address assigned to '1'B and the second contains the address assigned '0' B.
CHRTBL	Table into which character constants are entered.
CHAR2	Allocated address of the character constants in CHRTBL are put in this table.
IDAREA	Identifiers reside in this table for as long as they are required.
ATR	This array is used for passing the attributes to the symbol table handling routines as follows:
	ATR(1) Basic type (fixed/float/char...)

ATR(2) : Storage class (static/external...)

ATR(3) : No. of dimensions if a variable or, for labels whether or not the definition has been made.

ATR(4) : Procedure count

ATR(5) : Depth of procedure nesting.

ATR(6) : Not used

ATR(7) : Offset from respective data area bases

ATR(8) : Pointer to bounds area (for arrays)

SYMTA1 This array is part of the symbol table. For each entry in the symbol table there is a pointer in SYMTA1 which points to the relevant identifier in IDAREA

SYMTA2 This array is also part of the symbol table. It contains the first six attributes (as described in ATR) of the entry in packed form.

IP It is an array which contains the nonterminals on the left hand side of each production.

LSYM Pointer to the first empty cell in symbol table.

SYMSTK Stack for saving IBLPTR

IBLPTR Pointer of the current block into SYMTA1 .

OP An array of opcodes used by quadruple generator.

APPENDIX C

SAMPLE MINIPL SOURCE PROGRAM LISTING

Source Program Input to the MINIPL Compiler

```
TEST ..   PROCERURE OPTIONS (MAIN),.DECLARE
A FIXED, B FLOAT, C CHAR, D FIXED,
    GET LIST (A,B,C,D);.
    A = B,.
    IF A = D  THEN PUT LIST (C,B),. ELSE
  A=A/D,. BEGIN,. DECLARE E FLOAT,.
    E=2.0E+3,. B=E+B,.
    DO WHILE B NE E,. E = E+1,. END,.
    B=E*B,. END,.END,.
```

Indented Listing Produced by the Compiler

```
TEST .. PROCEDURE OPTIONS (MAIN),.
DECLARE   A FIXED, B FLOAT, C CHAR, D FIXED,.
GET LIST (A,B,C,D),.
A = B,.
IF A=D THEN
    PUT LIST (C,B),.
ELSE A = A/D,.
    BEGIN,.
    DECLARE E FLOAT,.
    E=2.0E+3,.
    B=E+B,.
      DO WHILE B NE E,.
      E=E+1,.
      END,.
    B=E*B,.
    END,.
 END,..
```

APPENDIX D

LIST OF ERROR MESSAGES

1. WAITING FOR MAIN/EXTERNAL PROCEDURE

2 NESTING TOO DEEP. PROGRAM TERMINATED

3 ELSE BY ITSELF IS MEANINGLESS

4 A STMT SHOULD COME BEFORE 'ELSE'

5 SUPERFLUOUS END

6 PROCEDURE AT THE WRONG PLACE

7 SECOND PROGRAM STARTS BEFORE THE FIRST ENDED

8 DECLARATION SHOULD BE THE FIRST THING IN A BEGIN OR PROCEDURE BLOCK

9 NESTING TOO DEEP. OLD MARGIN RESUMED

10 THERE SHOULD BE A DELIMITER ON EITHER SIDE OF A STRING CONSTANT

11 ILLEGAL TERMINAL .... (The terminal is printed here)

14 SYMBOL TABLE OVERFLOW. JOB TERMINATED

15 IDAREA OVERFLOW. JOB TERMINATED

16 GROUP COUNT MISSING

APPENDIX E

A SAMPLE PROGRAM FOR QUADRUPLE PROCESSING

```
ADDFIX MACRO   A1,A2,B1,B2,C1,C2
       IFT     A1=100
       CLA     S.&A2
       IFT     A1=101
       CLA     A2
       IFT     A1=100
       DUP     9,0
       IFT     A1=101
       DUP     7,0
       IFT     A1=102                 CHECK IF SAME DEPTH
       CLA     A2,1
       IFT     A1=102
       DUP     3,0
       STO     ADTEMP
       CLA     A1+1,1
       PAC     ,2
       CLA     A2,2
       CLA     ADTEMP
       IFT     B1=100
       ADD     S.&B2
       IFT     B1=101
       ADD     B2
       IFT     B1=100
       DUP     9,0
       IFT     B1=101
       DUP     7,0
       IFT     B1=102                 CHECK IF SAME DEPTH
       ADD     B2,1
       IFT     B1=102
       DUP     3,0
       STO     ADTEMP
       CLA     B1+1,1
       PAC     ,2
       ADD     B2,2
       CLA     ADTEMP
       IFT     C1=100
       STO     S.&C2
       IFT     C1=101
       STO     C2
       IFT     C1=100
       DUP     9,0
       IFT     C1=101
       DUP     7,0
       IFT     C1=102
```

```
STO      C2,1
IFT      C1=102
DUP      3,0
STO      ADTEMP
CLA      C1+1,1
PAC      ,2
STO      C2,2
CLA      ADTEMP
ENDM
```

APPENDIX F

STORAGE MANAGEMENT MACROS

```
SETREA MACRO     REQMNT
*       THIS MACRO SETS THE DYNAMIC SPACE FOR THE MAIN PROCEDURE
*  INDEX REGISTER   IS USED TO HOLD THE BASE ADDR. OF CURRENTLY ACTIVE
*       PROCEDURE
*       ALL ADDRESSES IN STACK ARE TRUE ADDRESSES AND IN I.R. ARE COMPL.
*       REQMNT    IS THE DYNAMIC AREA REQUIRE MENT FOR THE MAIN PROC.
        AXT       STACK-1,1
        PXA       ,1
        STA       STACK     FIRST LOCN OF MAIN PROCEDURE AREA NOW POINTS
*                              TO ITS OWN BOTTOM(ONE BELOW THE FIRST CELL)
        PAC       ,1     I.REG. 1 NOW POINTS TO STACK BOTTOM
        ADD       =REQMNT    STKTOP NOEW POINTS TO THE TOP OF STACK
        ENDM
```

```
FRIREA MACRO     I
*   I IS THE LEVEL OF THE ROUTINE BEING EXITED
*  THIS ROUTINE DOES ALL THE EXIT FORMALITIES. ARG. RETURN TO BE INCLUD.
        CLA      I+2,1
        PAC      ,4       RETURN ADDRESS IS IN I.R.4 NOW.
        CLA      I+3,1
        STA      ADTEMP
        PXA      ,1
        PAC      ,1
        SXA      STKTOP        STKTOP SET TO PREVIOUS(EXITED) PROCBOTM.
        LAC      ADTEMP,1       REGISTET 1 SET TO OLD PROCEDURE.BASE ADR
        TRA      1,4
        ENDM
```

```
GETREA MACRO      I,REQMNT
*  GETREA IS CALLED WHEN ANY PROCEDURE IS ENTERED
*    I IS THE LEVEL OF THE INVOKED PROCEDURE
*  REQMNT IS THE DYNAMIC STORE REQUIREMENT OF THE INVOKED PROCEDURE
*       RETURN ADDRESS IS STILL IN I.REG. 4
        LAC      STKTOP,      I.R. 2 POINTS TO TOP OF STACK(FUTURE BOTTOM)
        PXA     ,4
        PAC     ,4
        PXA     ,4
        STA      I+2,2      I+2   &TH LOCN FROM BOTTOM TC CONTAIN RETURN A
        PXA     ,1
        PAC     ,4
        PXA     ,4
        STA      I+3,2      BASE ADDR. OF CALLING PROC. IN I+3&TH LOCN
 *      COPY THE FIRST I  LOCNS OF PREV DATA PAREA INTC THE NEW ONE.
 *      THESE ARE THE BASE ADDRESSES OF THE REFERENCABLE PARENTS
        AXT      1,4
COPY    CLA      1,1
        STA      1,2
        TXI      *+1,1,1    INCREMENT TO POINT TO NEXT LOCN.
        TXI      *+1,2,1    INCREMENT TO POINT TO NEXT LOCN.
        TXI      *+1,4,1 COUNTER,I.R.4) INCREMENTS.
        TXL      COPY,4,I
        LAC      STKTOP,1
        CLA      STKTOP
        ADD      =REQMNT
        STO      STKTOP
        ENDM
```

APPENDIX G

LIST OF QUADRUPLES

A list of quadruples is given. Where necessary a comment is given otherwise explanation is to be found in Chapter 7 (Part II) or is not needed. In the quadruples for arithmetic operations F stands for floating point operation (as in ADPF) and I for integer operations. In the list, ad1, ad2 and ad3 are the addresses of locations, label is a string of digits which will be used to refer to a label (e.g. 77 will refer to label S.77); and integer gives a directive to the quadruple processor.

ADDF	ad1,	ad2,	ad3
ADDI	ad1,	ad2,	ad3
SUBF	ad1,	ad2,	ad3
SUBI	ad1,	ad2,	ad3
DIVF	ad1,	ad2,	ad3
DIVI	ad1,	ad2,	ad3
MULF	ad1,	ad2,	ad3
MULI	ad1,	ad2,	ad3
OR	ad1,	ad2,	ad3
AND	ad1,	ad2,	ad3
NOT	ad1,	ad2,	ad3

```
RELINT    ad1,  ad2,  ad3       Convert real to integer
CHRINT    ad1,  ad2,  ad3       Convert character to integer
INTCHR    ad1,  ad2,  ad3       Convert integer to character
BOLINT    ad1,  ad2,  ad3       Convert bool. to integer
BOLREL    ad1,  ad2,  ad3       Convert bool. to real
BOLCHR    ad1,  ad2,  ad3       Convert bool. to char.
INTBOL    ad1,  ad2,  ad3       Convert int. to bool
RELBOL    ad1,  ad2,  ad3       Convert real to bool
CHRBOL    ad1,  ad2,  ad3       Convert char. to bool
RELCHR    ad1,  ad2,  ad3       Convert real to char.
CHRREL    ad1,  ad2,  ad3       Convert char. to real
INTREL    ad1,  ad2,  ad3       Convert integer to real
LT        ad1,  ad2,  ad3
GT        ad1,  ad2,  ad3
EQ        ad1,  ad2,  ad3
LE        ad1,  ad2,  ad3
GE        ad1,  ad2,  ad3
NE        ad1,  ad2,  ad3
NG        ad1,  ad2,  ad3
NL        ad1,  ad2,  ad3
READIN
FILRD
FMTADR    label
FMTLSS
GETLST    ad1,  ad2
```

DOIO	
STOR	ad1
ENDIO	
GOTO	label
BSSS	integer
RITPRN	
CLAD	ad1
PUTLST	ad1
FILPR	
PROUT	
STRING	
ADR	ad1
PAGE	
SKIP	ad1
FLTINT	ad1
FTYPE	integer, ad2
FTYPE2	ad1, ad_2
LFTPRN	
GRPCNT	ad1
SKPFMT	ad1